
Title of Workshop
Intel® Xeon Phi™ Product Family
Intel ® MPI
Klaus-Dieter Oertel, July 4th 2013
Thanh Phung (presenter)
Software and Services Group
Intel Corporation

Legal Disclaimer

- INFORMATION IN THIS DOCUMENT IS PROVIDED IN CONNECTION WITH INTEL® PRODUCTS. NO LICENSE, EXPRESS OR IMPLIED, BY ESTOPPEL OR OTHERWISE, TO ANY INTELLECTUAL PROPETY RIGHTS IS GRANTED BY THIS DOCUMENT. EXCEPT AS PROVIDED IN INTEL'S TERMS AND CONDITIONS OF SALE FOR SUCH PRODUCTS, INTEL ASSUMES NO LIABILITY WHATSOEVER, AND INTEL DISCLAIMS ANY EXPRESS OR IMPLIED WARRANTY, RELATING TO SALE AND/OR USE OF INTEL® PRODUCTS INCLUDING LIABILITY OR WARRANTIES RELATING TO FITNESS FOR A PARTICULAR PURPOSE, MERCHANTABILITY, OR INFRINGEMENT OF ANY PATENT, COPYRIGHT OR OTHER INTELLECTUAL PROPERTY RIGHT.
- Intel may make changes to specifications and product descriptions at any time, without notice.
- All products, dates, and figures specified are preliminary based on current expectations, and are subject to change without notice.
- Intel, processors, chipsets, and desktop boards may contain design defects or errors known as errata, which may cause the product to deviate from published specifications. Current characterized errata are available on request.
- Sandy Bridge and other code names featured are used internally within Intel to identify products that are in development and not yet publicly announced for release. Customers, licensees and other third parties are not authorized by Intel to use code names in advertising, promotion or marketing of any product or services and any such use of Intel's internal code names is at the sole risk of the user
- Software and workloads used in performance tests may have been optimized for performance only on Intel microprocessors. Performance tests, such as SYSmark and MobileMark, are measured using specific computer systems, components, software, operations and functions. Any change to any of those factors may cause the results to vary. You should consult other information and performance tests to assist you in fully evaluating your contemplated purchases, including the performance of that product when combined with other products. For more information go to http://www.intel.com/performance
- Intel, Core, Xeon, VTune, Cilk, Intel and Intel Sponsors of Tomorrow. and Intel Sponsors of Tomorrow. logo, and the Intel logo are trademarks of Intel Corporation in the United States and other countries.
- *Other names and brands may be claimed as the property of others.
- Copyright ©2011 Intel Corporation.
- Hyper-Threading Technology: Requires an Intel® HT Technology enabled system, check with your PC manufacturer. Performance will vary depending on the specific hardware and software used. Not available on all Intel® Core™ processors. For more information including details on which processors support HT Technology, visit http://www.intel.com/info/hyperthreading
- Intel® 64 architecture: Requires a system with a 64-bit enabled processor, chipset, BIOS and software. Performance will vary depending on the specific hardware and software you use. Consult your PC manufacturer for more information. For more information, visit http://www.intel.com/info/em64t
- Intel® Turbo Boost Technology: Requires a system with Intel® Turbo Boost Technology capability. Consult your PC manufacturer. Performance varies depending on hardware, software and system configuration. For more information, visit http://www.intel.com/technology/turboboost

Objectives

- Intel ® MPI execution models on Intel® Xeon Phi™
- Pure MPI or hybrid MPI applications on Intel® Xeon Phi™
- Analysis of Intel® MPI codes with the Intel ® Trace Analyzer and Collector (ITAC) on Intel® Xeon Phi™
- Load balancing on heterogenous systems
- Debugging Intel ® MPI codes on Intel® Xeon Phi™

Outline

- Overview
- Prerequisites of Intel® MPI
- Programming Models
- Hybrid Computing
- Intel® Trace Analyzer and Collector
- Load Balancing
- Debugging

Outline

- Overview
- Prerequisites of Intel® MPI
- Programming Models
- Hybrid Computing
- Intel® Trace Analyzer and Collector
- Load Balancing
- Debugging

Enabling & Advancing Parallelism

High Performance Parallel Programming

Intel tools, libraries and parallel models extend to multicore, many-core and heterogeneous computing

Code

Compiler
Libraries
Parallel Models

intel Software

Multicore

Many-core

Cluster

Multicore CPU

Multicore CPU

Intel® Xeon Phi™ Coprocessor

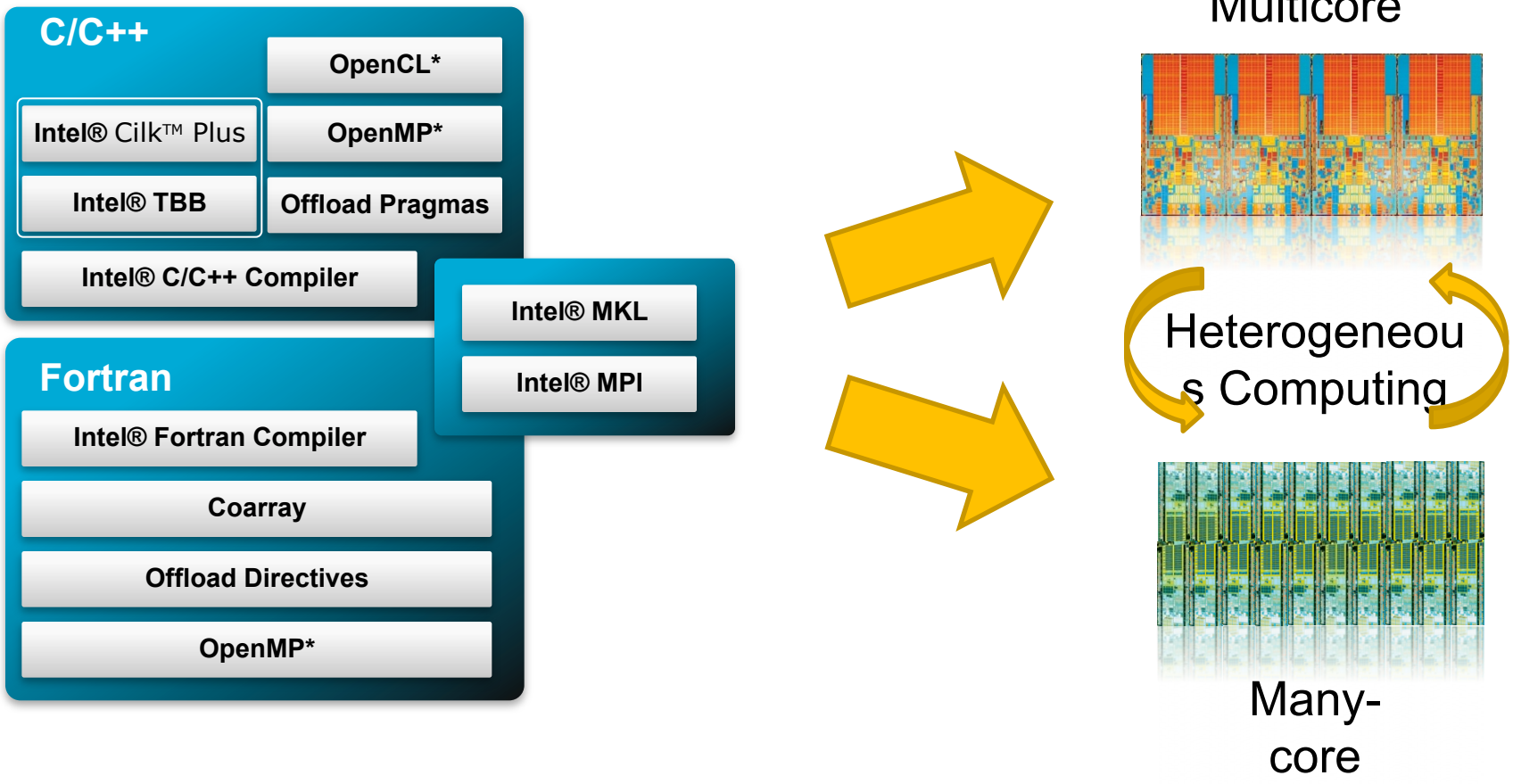
Multicore Cluster

Multicore & Many-core Cluster

Use One Software Architecture Today. Scale Forward Tomorrow.

Preserve Your Development Investment

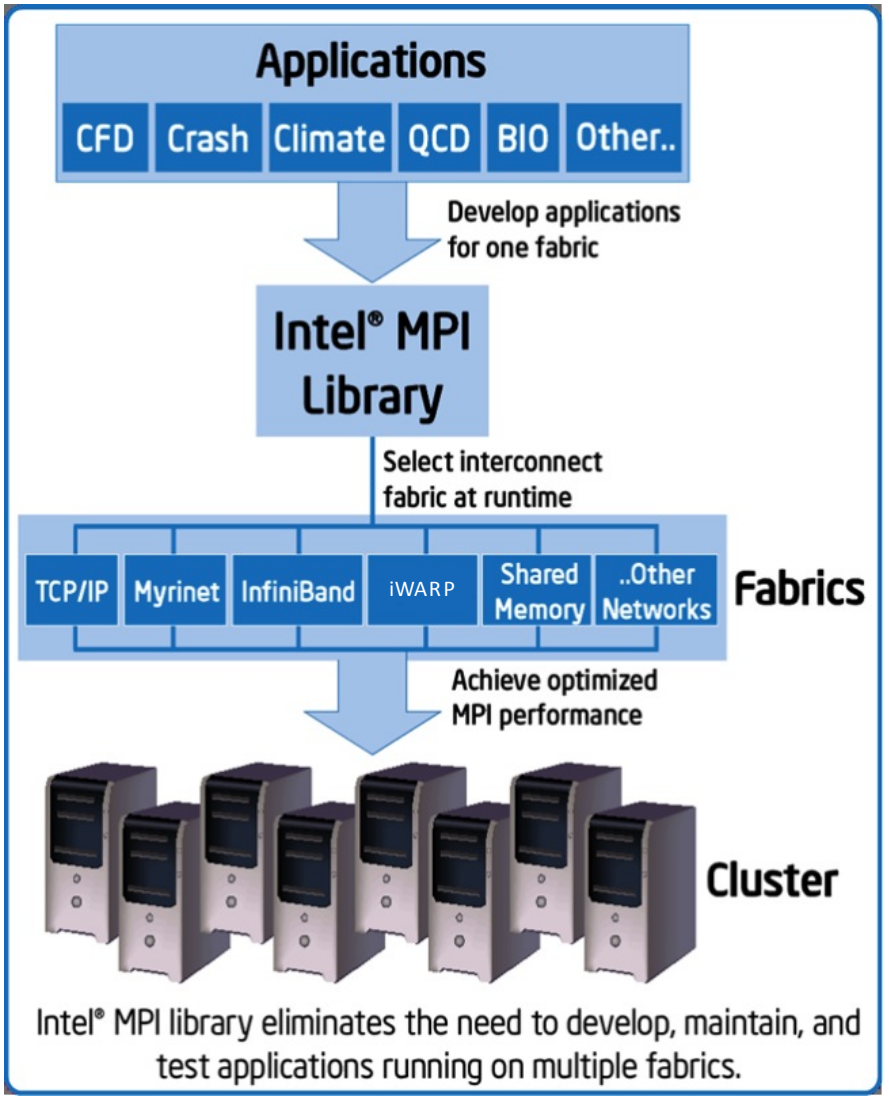
Common Tools and Programming Models for Parallelism



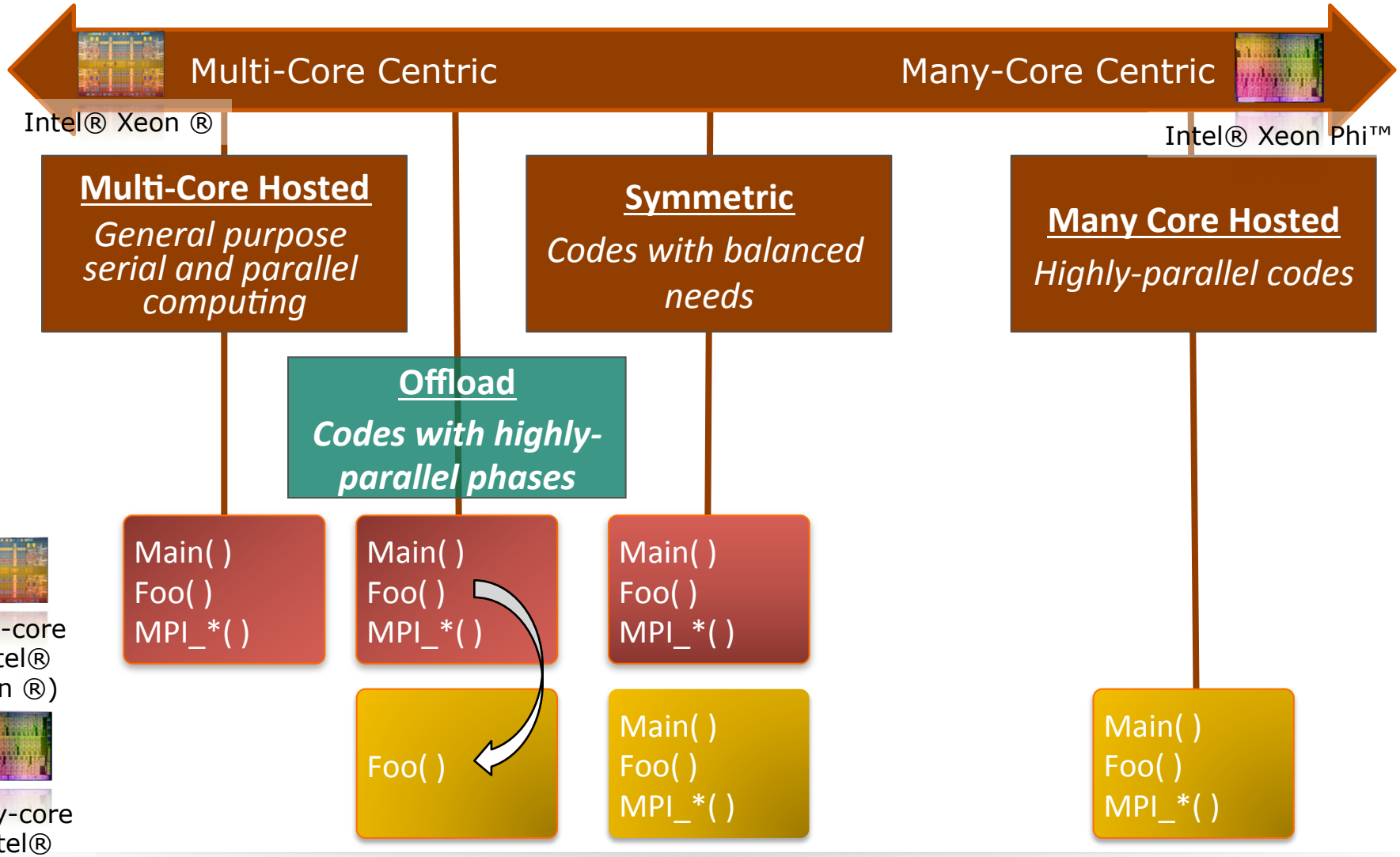
Develop Using Parallel Models that Support Heterogeneous Computing

Intel® MPI Library Overview

- Intel is a leading vendor of MPI implementations and tools
- Optimized MPI application performance
  - Application-specific tuning
  - Automatic tuning
  - Low latency
  - Industry leading latency
- Interconnect Independence & Runtime Selection
  - Multi-vendor interoperability
  - Performance optimized support for the latest OFED capabilities through DAPL 2.0
- More robust MPI applications
  - Seamless interoperability with Intel® Trace Analyzer and Collector



Spectrum of Programming Models and Mindsets

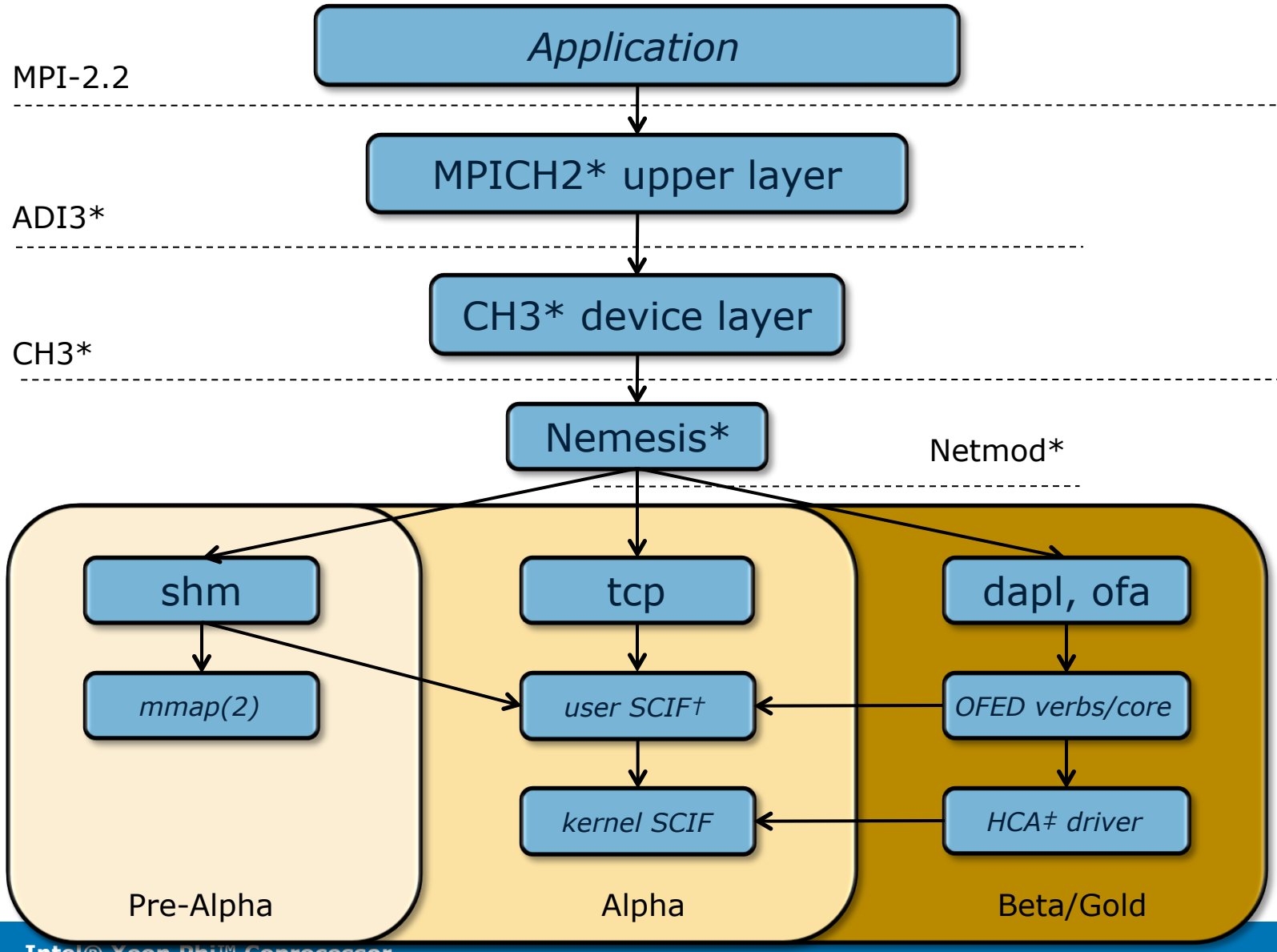


Range of models to meet application needs

Levels of communication speed

- Current clusters are not homogenous regarding communication speed:
  - Inter node (InfiniBand*, Ethernet, etc)
  - Intra node
    - Inter sockets (Quick Path Interconnect)
    - Intra socket
- Two additional levels to come with Intel® Xeon Phi™ coprocessor:
  - Host-coprocessor communication
  - Inter coprocessor communication

Intel® MPI Library Architecture & Staging



Selecting network fabrics

- Intel® MPI automatically selects the best available network fabric it can find.
  - Use `I_MPI_FABRICS` to select a different communication device explicitly
- The best fabric is usually based on InfiniBand* (dapl, ofa) for inter node communication and shared memory for intra node
- Available for Intel® Xeon Phi™ coprocessor:
  - `shm, tcp, ofa, dapl`
  - Availability checked in the order `shm:dapl, shm:ofa, shm:tcp` (intra:inter)

Outline

- Overview
- Prerequisites of Intel® MPI
- Programming Models
- Hybrid Computing
- Intel® Trace Analyzer and Collector
- Load Balancing
- Debugging

Installation

- Download latest Intel® MPI and plus a license from Intel Registration Center (included in Intel® Cluster Studio XE)
  - `l_mpi_p_4.1.1.036.tgz` (later: `l_itac_p_8.1.2.033.tgz`)
- Unpack the tar file, and execute the installation script:

```
# tar zxf l_mpi_p_4.1.1.036.tgz
# cd l_mpi_p_4.1.1.036
# ./install.sh
```

  - Follow the installation instructions
- Root or user installation possible!
- Resulting directory structure has intel64 and mic sub-dirs.:

```
/opt/intel/impi/4.1.1.036/intel64/{bin,etc,include,lib}
/opt/intel/impi/4.1.1.036/mic/{bin,etc,include,lib}
```

  - Only one user environment setup required, serves both architectures!

Prerequisites

- Make sure the Intel® Xeon Phi™ coprocessor is accessible through the network
- Assumption: Hostname `host-mic0` is associated to IP address
  - Specified in `/etc/hosts` or `$HOME/.ssh/config`
- The tools directory /opt/intel is mounted by NFS onto the coprocessor

- If NFS is not available: Upload Intel® MPI libraries onto the coprocessor(s)

```
# cd /opt/intel/impi/4.1.1.036/mic/lib
  scp libmpi.so.4.1 host-mic0:/lib64/libmpi.so.4
  ...
```

  - Execute as root or user with sudo rights (if not possible, copy to user directory)
  - Has to be repeated after every re-boot of the coprocessor!

Prerequisites per User

- Set the compiler environment
  ```
  # source <compiler_installdir>/bin/compilervars.sh intel64
  ```
  - Identical for Host and coprocessor
- Set the Intel® MPI environment
  ```
  # source /opt/intel/impi/4.1.1.036/intel64/bin/mpivars.sh
  ```
  - Identical for Host and coprocessor

- Enable Intel® MPI execution on the coprocessor
  ```
  # export I_MPI_MIC=enable
  ```
  - Set on the host to start Intel® MPI processes on the coprocessor!

- mpirun needs ssh access to the Intel® Xeon Phi™ coprocessor!
  - Done! User's ssh key `~/.ssh/id_rsa.pub` is copied to the coprocessor at driver boot time.

Compiling and Linking for Intel® Xeon Phi™ Coprocessor

- Compile MPI sources using Intel® MPI scripts (C/C++)
  - For the host including any potential offload code

    ```
    # mpiicc -o test test.c
    ```

    - Use flag `-no-offload` to suppress potential offload code
  - For native execution on the coprocessor add „–mmic“ flag, i.e. the usual compiler flag controls also the MPI compilation

    ```
    # mpiicc -mmic -o test test.c
    ```
- Linker verbose mode “-v” shows
  - Without „–mmic“ linkage with intel64 libraries:

    ```
    ld ... -L/opt/intel/impi/4.1.1.036/intel64/lib ...
    ```
  - With „–mmic“ linkage with coprocessor libraries:

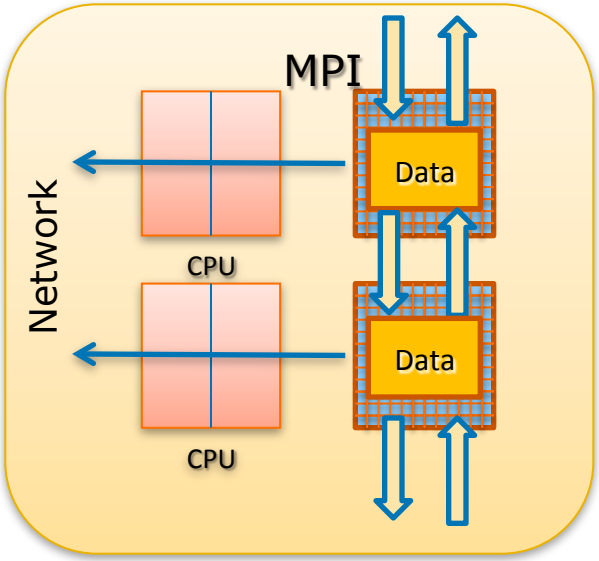
    ```
    ld ... -L/opt/intel/impi/4.1.1.036/mic/lib ...
    ```

Outline

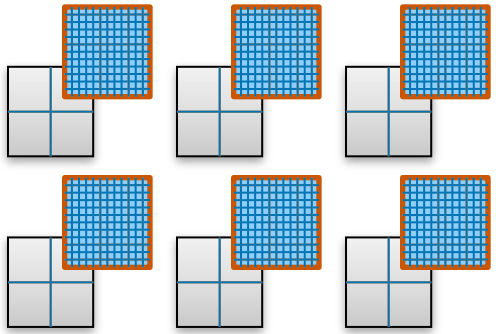
- Overview
- Prerequisites of Intel® MPI
- Programming Models
- Hybrid Computing
- Intel® Trace Analyzer and Collector
- Load Balancing
- Debugging

Coprocessor-only Programming Model

- MPI ranks on Intel® Xeon Phi™ coprocessor(only)
- All messages into/out of coprocessors
- Intel® Cilk™ Plus, OpenMP*, Intel® Threading Building Blocks, Pthreads used directly within MPI processes



Homogenous network of many-core CPUs



Build Intel® Xeon Phi™ binary using the Intel® compiler.

Upload the binary to the Intel® Xeon Phi™ coprocessor.

Run instances of the MPI application on Intel® Xeon Phi™ coprocessor nodes.

Coprocessor-only Programming Model

- MPI ranks on the Intel® Xeon Phi™ coprocessor(s) only
- MPI messages into/out of the coprocessor(s)
- Threading possible

- Build the application for the Intel® Xeon Phi™ coprocessor

```
# mpiicc -mmic -o test_hello.MIC test.c
```

- Launch the application on the coprocessor from host

```
# export I_MPI_MIC=enable
# mpirun -n 2 -host host-mic0 ./test_hello.MIC
```

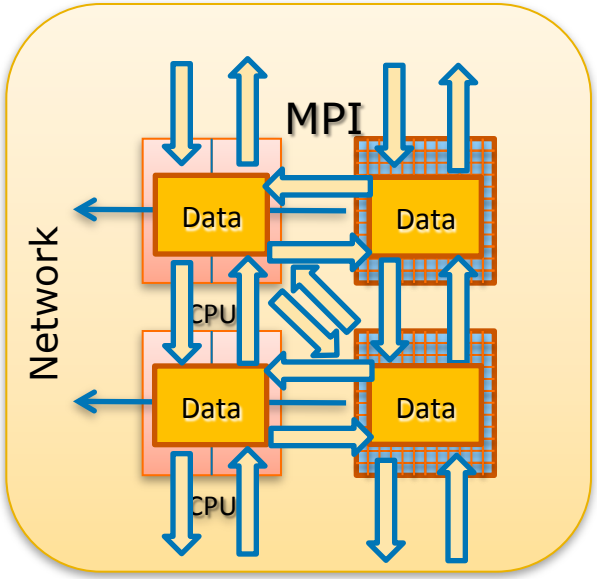
- Alternatively: login to the Intel® Xeon Phi™ coprocessor and start the MPI run there!
- No NFS: Upload the Intel® Xeon Phi™ executable and add working directory flag

```
# scp ./test_hello.MIC host-mic0:/my_mic_dir/
# mpirun … -wdir /my_mic_dir/ …
```

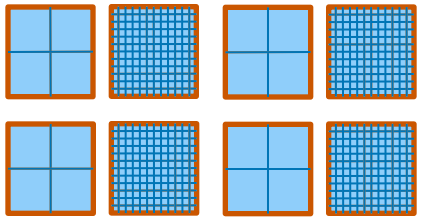

20

Symmetric Programming Model

- MPI ranks on Intel® Xeon Phi™ Architecture and host CPUs
- Messages to/from any core
- Intel® Cilk™ Plus, OpenMP*, Intel® Threading Building Blocks, Pthreads* used directly within MPI processes



Heterogeneous network of homogeneous CPUs



Build binaries by using the resp. compilers targeting Intel® 64 and Intel® Xeon Phi™ Architecture.

Upload the binary to the Intel® Xeon Phi™ coprocessor.

Run instances of the MPI application on different mixed nodes.

Symmetric model

- MPI ranks on the coprocessor(s) and host CPU(s)
- MPI messages into/out of the coprocessor(s) and host CPU(s)
- Threading possible

- Build the application for Intel®64 and the Intel® Xeon Phi™ Architecture separately

```
# mpiicc -o test_hello test.c
# mpiicc -mmic -o test_hello.MIC test.c
```

- Launch the application on the host and the coprocessor

```
# export I_MPI_MIC=enable
# mpirun -n 2 -host <hostname> ./test_hello :
         -n 2 -host host-mic0 ./test_hello.MIC
```

- No NFS: Upload the Intel® Xeon Phi™ executable and add flag

```
# scp ./test_hello.MIC host-mic0:/my_mic_dir/
# mpirun … : -wdir /my_mic_dir/ …
```


22

Utilize the POSTFIX env variable
Support for NFS-shared cards

- Assumption: The current working directory is available with identical path on the coprocessor (e.g. mounted)
- Specify the suffix of the coprocessor binary

```
# export I_MPI_MIC_POSTFIX=.MIC
```

- Specify the node names in a file

```
# cat mpi_hosts
  host
  host-mic0
```

- Execute in symmetric mode on the host and the coprocessor

```
# export I_MPI_MIC=enable
# mpirun –f mpi_hosts –n 4 ./test_hello
```

- The binary `./test_hello${I_MPI_MIC_POSTFIX}` will be used by mpirun on the coprocessor

Utilize the PREFIX env variable
Support for NFS-shared cards

- Assumption: The current working directory is available with identical path on the coprocessor (e.g. mounted)
- Place the coprocessor binary in a separate directory, but with identical basename of the host

```
# mpiicc -mmic -o ./MIC/test_hello test.c
```

- Specify the prefix of the coprocessor binary

```
# export I_MPI_MIC_PREFIX=./MIC/
```

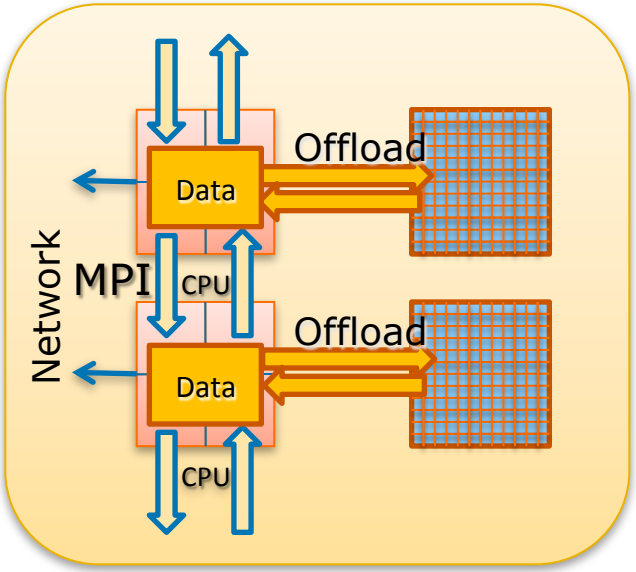
- Execute in symmetric mode on the host and the coprocessor

```
# export I_MPI_MIC=enable
# mpirun -f mpi_hosts -n 4 ./test_hello
```

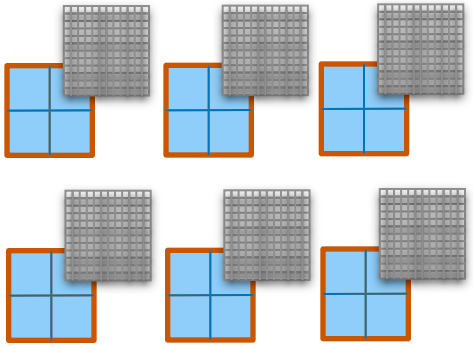
- The binary `${I_MPI_MIC_PREFIX}./test_hello` will be used by mpirun on the coprocessor

MPI+Offload Programming Model

- MPI ranks on Intel® Xeon® processors (only)
- All messages into/out of host CPUs
- Offload models used to accelerate MPI ranks
- Intel® Cilk™ Plus, OpenMP*, Intel® Threading Building Blocks, Pthreads* within Intel® Xeon Phi™ coprocessors



Homogenous network of heterogeneous nodes



Build Intel® 64 executable with included offload by using the Intel® compiler.

Run instances of the MPI application on the host, offloading code onto the coprocessor.

Advantages of more cores and wider SIMD for certain applications

MPI+Offload Programming Model

- MPI ranks on the host CPUs only
- MPI messages into/out of the host CPUs
- Intel® Xeon Phi™ coprocessor as an accelerator

- Compile for MPI and internal offload

```
# mpiicc -o test test.c
```

- Latest compiler compiles by default for offloading if offload construct is detected!
  - Switch off by `-no-offload` flag
- Execute on host(s) as usual

```
# mpirun -n 2 ./test
```

- MPI processes will offload code for acceleration

26

Offloading to Intel® Intel® Xeon Phi™ Architecture

C/C++ Offload Pragma

```
#pragma offload target (mic)
#pragma omp parallel for reduction(+:pi)
for (i=0; i<count; i++)  {
        float t = (float)((i+0.5)/count);
        pi += 4.0/(1.0+t*t);
}
pi /= count;
```

MKL Implicit Offload

```
//MKL implicit offload requires no source code
changes, simply link with the offload MKL Library.
```

MKL Explicit Offload

```
#pragma offload target (mic) \
     in(transa, transb, N, alpha, beta) \
     in(A:length(matrix_elements)) \
     in(B:length(matrix_elements)) \
     in(C:length(matrix_elements)) \
     out(C:length(matrix_elements)alloc_if(0))
     sgemm(&transa, &transb, &N, &N, &N, &alpha,
           A, &N, B, &N, &beta, C, &N);
```

Fortran Offload Directive

```
!dir$ omp offload target(mic)
!$omp parallel do
      do i=1,10
      A(i) = B(i) * C(i)
      enddo
!$omp end parallel
```

C/C++ Language Extensions

```
class _Shared common  {
         int data1;
         char *data2;
         class common *next;
         void process();
};
_Shared class common  obj1, obj2;

…
_Cilk_spawn _Offload obj1.process();
_Cilk_spawn          obj2.process();

…
```

Outline

- Overview
- Prerequisites of Intel® MPI
- Programming Models
- Hybrid Computing
- Intel® Trace Analyzer and Collector
- Load Balancing
- Debugging

Traditional Cluster Computing

- MPI is »the« portable cluster solution
- Parallel programs use MPI over cores inside the nodes
  - Homogeneous programming model
  - "Easily" portable to different sizes of clusters
  - No threading issues like »False Sharing« (common cache line)
  - Maintenance costs only for one parallelization model

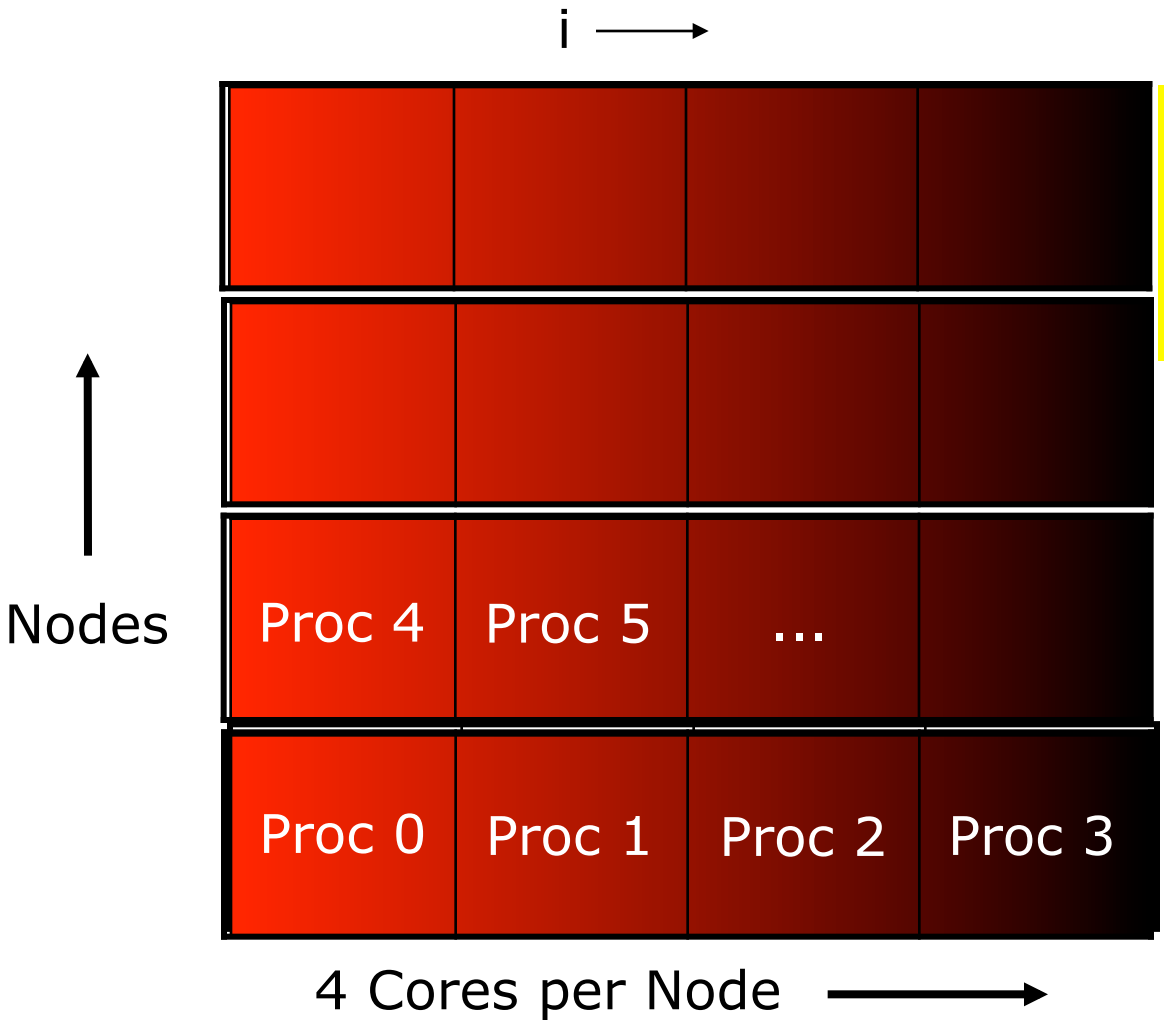
Traditional Cluster Computing (contd.)

- Hardware trends
  - Increasing number of cores per node - plus cores on co-processors
  - Increasing number of nodes per cluster
- Consequence: Increasing number of MPI processes per application
- Potential MPI limitations
  - Memory consumption per MPI process, sum exceeds the node memory
  - Limited scalability due to exhausted interconnects (e.g. MPI collectives)
  - Load balancing is often challenging in MPI

Hybrid Computing

- Combine MPI programming model with threading model
- Overcome MPI limitations by adding threading:
  - Potential memory gains in threaded code
  - Better scalability (e.g. less MPI communication)
  - Threading offers smart load balancing strategies
- Result: Maximize performance by exploitation of hardware (incl. co-processors)

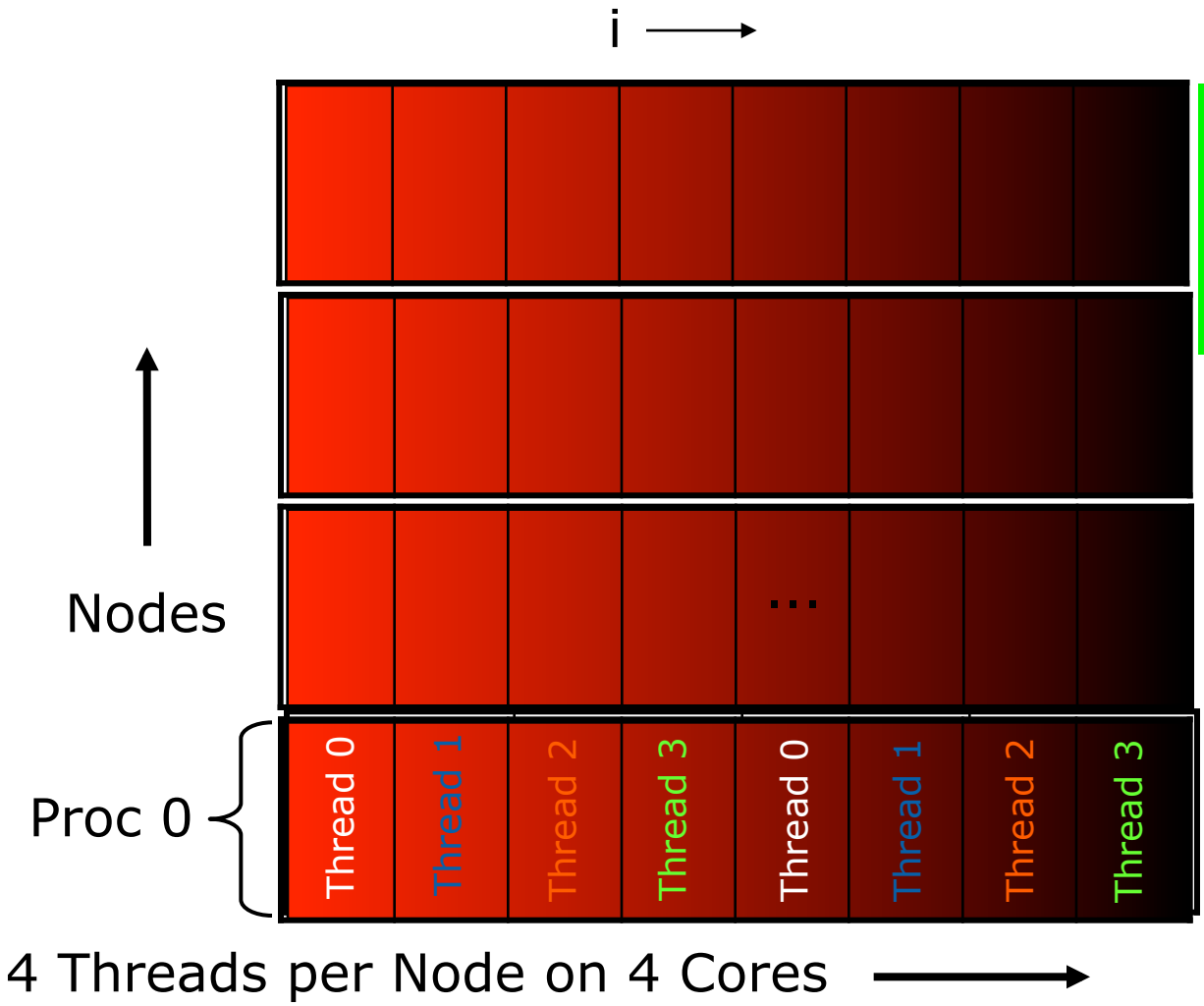
Example: MPI Load Imbalance



Difficult to implement load balancing in nodes with MPI

Dark red = high load

Example: Hybrid Load Balance

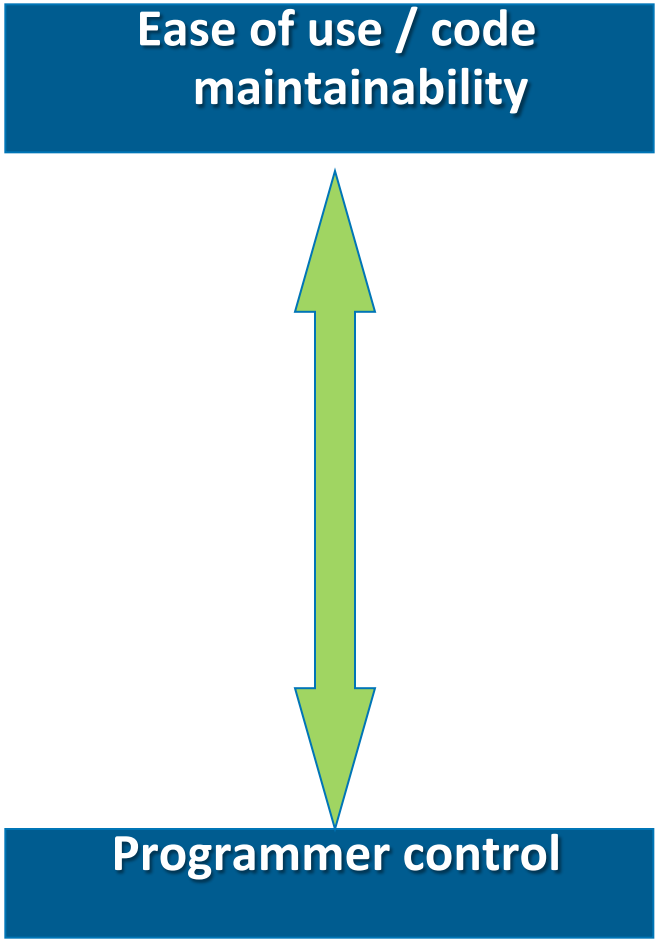


Interleaved OpenMP threads improve total load balancing

Dark red = high load

Options for Thread Parallelism

- Intel® Math Kernel Library
- OpenMP*
- Intel® Threading Building Blocks
  Intel® Cilk™ Plus
- Pthreads* and other threading libraries



Choice of unified programming to target Intel® Xeon and Intel® Xeon Phi™!

Intel® MPI Support of Hybrid Codes

- Intel® MPI is strong in mapping control
- Sophisticated default or user controlled
  - `I_MPI_PIN_PROCESSOR_LIST` for pure MPI
  - For hybrid codes (takes precedence):

    `I_MPI_PIN_DOMAIN` =<*size*>[:<*layout*>]

    <size> =

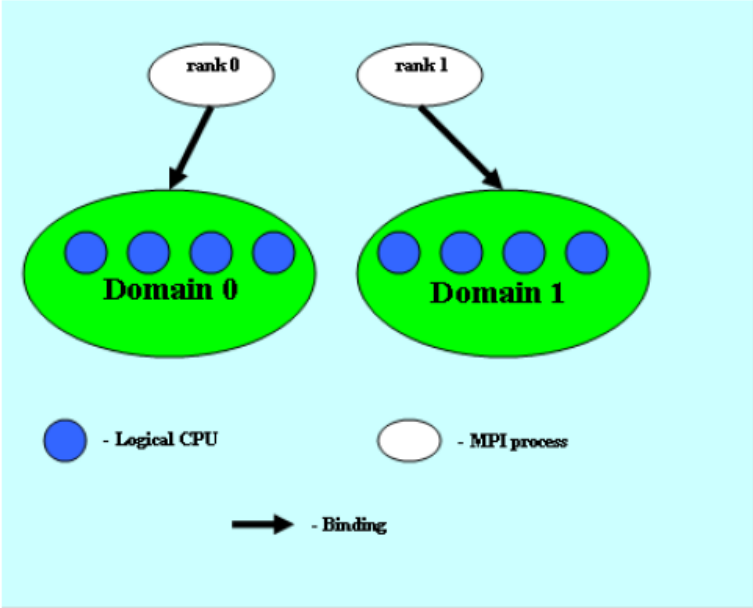
`omp`	Adjust to OMP_NUM_THREADS
`auto`	#CPUs/#MPIprocs
`<n>`	Number

    <layout> =

`platform`	According to BIOS numbering
`compact`	Close to each other
`scatter`	Far away from each other
- Naturally extends to hybrid codes on Intel® Xeon Phi™

Intel® MPI Support of Hybrid Codes

- Define `I_MPI_PIN_DOMAIN` to split logical processors into non-overlapping subsets
- Mapping rule: **1 MPI process per 1 domain**



Pin OpenMP threads inside the domain with `KMP_AFFINITY` (or in the code)

Intel® MPI Environment Support

- The execution command mpirun of Intel® MPI reads argument sets from the command line:
  - Sections between „:“ define an argument set (alternatively a line in a configfile specifies a set)
  - Host, number of nodes, but also environment can be set independently in each argument set

```
# mpirun -env I_MPI_PIN_DOMAIN 4 -host myXEON ...
        : -env I_MPI_PIN_DOMAIN 16 -host myMIC
```

- Adapt the important environment variables to the architecture
  - **OMP_NUM_THREADS**, **KMP_AFFINITY** for OpenMP
  - **CILK_NWORKERS** for Intel® Cilk™ Plus

Coprocessor-only and Symmetric Support

- Full hybrid support on Intel® Xeon from Intel ® MPI extends to the Intel ® Xeon Phi™ coprocessor
- **KMP_AFFINITY=balanced** (only on the coprocessor) in addition to **scatter** and **compact**
- **KMP_PLACE_THREADS=<n>Cx<m>T,<o>O** (**<n>**-Cores times **<m>**-Threads with **<o>**-cores Offset, only on coprocessor) in addition to **KMP_AFFINITY** for exact but still generic thread placement
- Recommendations:
  - Explicitly control where MPI processes and threads run in a hybrid application (according to threading model)
  - Avoid splitting cores among MPI processes, i.e. **I_MPI_PIN_DOMAIN** should be a multiple of 4
  - Try different **KMP_AFFINITY** and/or **KMP_PLACE_THREADS** settings for your application

OS Thread Affinity Mapping

- The Intel® Xeon Phi™ coprocessor has N cores, each with 4 hardware thread contexts, for a total of M=4*N threads
- The OS maps "procs" to the M hardware threads:

MIC core	0				1		…	(N-2)	(N-1)			
MIC HW thread	0	1	2	3	0	1	…	3	0	1	2	3
OS "proc"	1	2	3	4	5	6	…	(M-4)	0	(M-3)	(M-2)	(M-1)

- *The OS runs on proc 0, which lives on core (N-1)!*
  - Rule of thumb: Avoid using OS procs 0, (M-3), (M-2), and (M-1) to avoid contention with the OS
    - Only less than 2% resources unused (1/#cores)
  - Especially important when using the offload model due to data transfer activity!
  - But: Non-offload applications may slightly benefit from running on core (N-1)

OS Thread Affinity Mapping (ctd.)

- OpenMP library maps to the OS "procs"
- Examples (for non-offload apps which benefit from core N-1):
  - `KMP_AFFINITY=compact,granularity=thread,compact`

MIC core	0				1		…	(N-2)	(N-1)			
MIC HW thread	0	1	2	3	0	1	…	3	0	1	2	3
OS "proc"	1	2	3	4	5	6	…	(M-4)	0	(M-3)	(M-2)	(M-1)
OpenMP thread	***0***	***1***	***2***	***3***	4	5	…	(M-5)	(M-4)	(M-3)	(M-2)	(M-1)

  - `KMP_AFFINITY=balanced,granularity=thread OMP_NUM_THREADS=n=M/2`

MIC core	0				1		…	(N-2)	(N-1)			
MIC HW thread	0	1	2	3	0	1	…	3	0	1	2	3
OS "proc"	1	2	3	4	5	6	…	(M-4)	0	(M-3)	(M-2)	(M-1)
OpenMP thread	***0***	***1***			***2***	***3***	…		(n-2)	(n-1)		

OS Thread Affinity Mapping (ctd.)

- Examples (for non-offload apps which benefit from core N-1):
  - Default mapping is `KMP_AFFINITY=scatter,granularity=thread`

MIC core	0				1		…	(N-2)	(N-1)			
MIC HW thread	0	1	2	3	0	1	…	3	0	1	2	3
OS “proc”	1	2	3	4	5	6	…	(M-4)	0	(M-3)	(M-2)	(M-1)
OpenMP thread	0	N	2 N	3	1	5	…	(M-5)	(M-4)	(M-3)	(M-2)	(M-1)

MIC core	0				1		…	(N-2)	(N-1)			
MIC HW thread	0	1	2	3	0	1	…	3	0	1	2	3
OS “proc”	1	2	3	4	5	6	…	(M-4)	0	(M-3)	(M-2)	(M-1)
OpenMP thread	***0***				***1***		…		(n-2)	(n-1)		

OS Thread Affinity Mapping (ctd.)

- Use `balanced` affinity to minimize False Sharing!
  - `KMP_PLACE_THREADS=2Cx2T,0O`
  - but still with implicit default mapping `scatter,granularity=thread`

MIC core	0				1		…	(N-2)	(N-1)			
MIC HW thread	0	1	2	3	0	1	…	3	0	1	2	3
OS “proc”	1	2	3	4	5	6	…	(M-4)	0	(M-3)	(M-2)	(M-1)
OpenMP thread	***0***	***2***			***1***	***3***						

  - `KMP_PLACE_THREADS=2Cx2T,0O` and `KMP_AFFINITY=balanced`

MIC core	0				1		…	(N-2)	(N-1)			
MIC HW thread	0	1	2	3	0	1	…	3	0	1	2	3
OS “proc”	1	2	3	4	5	6	…	(M-4)	0	(M-3)	(M-2)	(M-1)
OpenMP thread	***0***	***1***			***2***	***3***						

MPI+Offload Support

- How to control mapping of threads on the coprocessor?
  - How do I avoid that offload of first MPI process interferes with offload of second MPI process, i.e. by using identical cores/threads on the coprocessor?
  - Default: No special support (now). Offloads from MPI processes handled by system like offloads from independent processes (or users).
- Define thread affinity manually per single MPI process:

```
# export OMP_NUM_THREADS=8
# mpirun -env KMP_AFFINITY=proclist=[1-8],explicit -n 1
         -host myHost ./test_mpioffload :
         -env KMP_AFFINITY=proclist=[9-16],explicit -n 1
         -host myHost ./test_mpioffload : ...
```

MPI+Offload Support (ctd.)

- Alternative: Use **KMP_PLACE_THREADS** and Intel® MPI rank number **PMI_RANK** in a wrapper script:

```
# cat ./wrapoffload.sh
  cores=$(( (OMP_NUM_THREADS+3)/4 ))
  offset=$(( cores*PMI_RANK ))
  export KMP_PLACE_THREADS=${cores}Cx4T,${offset}O
  ./test_mpioffload

# export OMP_NUM_THREADS=8
# mpirun -n 4 -host myHost ./wrapoffload.sh
```

- The mapping will be:

```
MPI rank 0: KMP_PLACE_THREADS=2Cx4T,0O == [1-8]
MPI rank 1: KMP_PLACE_THREADS=2Cx4T,2O == [9-16]
```

Create a configuration file for ease-of-use

- Instead of editing a long and tedious command line:

```
# mpirun –genv I_MPI_DEBUG 3
    -host host      –n 4 -env OMP_NUM_THREADS 2 ./test :
    -host host-mic0 –n 6 -env OMP_NUM_THREADS 8 ./test.MIC :
    -host host-mic1 –n 6 -env OMP_NUM_THREADS 8 ./test.MIC
```

- Create a configuration file (which you can edit as needed)

```
# cat conf_file
      –genv I_MPI_DEBUG 3
      -host host      –n 4 -env OMP_NUM_THREADS 2 ./test
      -host host-mic0 –n 6 -env OMP_NUM_THREADS 8 ./test.MIC
      -host host-mic1 –n 6 -env OMP_NUM_THREADS 8 ./test.MIC
```

- And always run the same command line

```
# mpirun –configfile conf_file
```

Outline

- Overview
- Prerequisites of Intel® MPI
- Programming Models
- Hybrid Computing
- Intel® Trace Analyzer and Collector
- Load Balancing
- Debugging

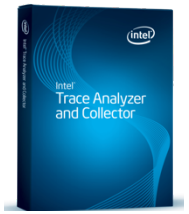
Introduction – What is Tracing?

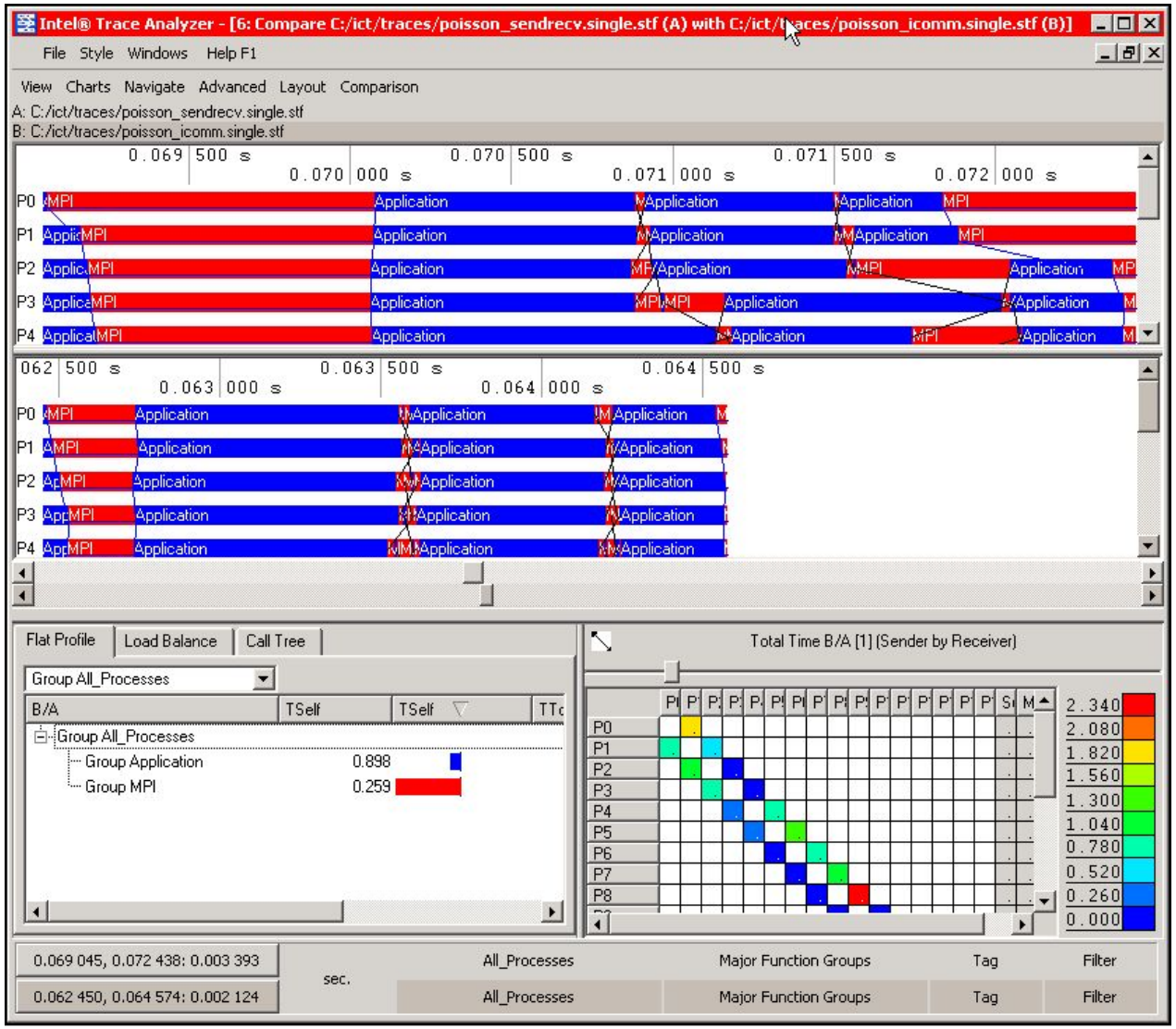
- Record program execution
  - Program events such as function enter/exit, communication
- 1:1 protocol of the actual program execution
  - Sampling gathers statistical information
- Accurate data
- Easily get loads of data

Event based approach

- Event = time stamp + thread ID + description
  - Function entry/exit
  - Messages
  - Collective operations
  - Counter samples
- Strengths:
  - Predict detailed program behavior
  - Record exact sequence of program states – keep timing consistent
  - Collect information about exchange of messages: at what times and in which order
  - Detect temporal dependencies

Intel® Trace Analyzer and Collector





Compare the event timelines of two communication profiles

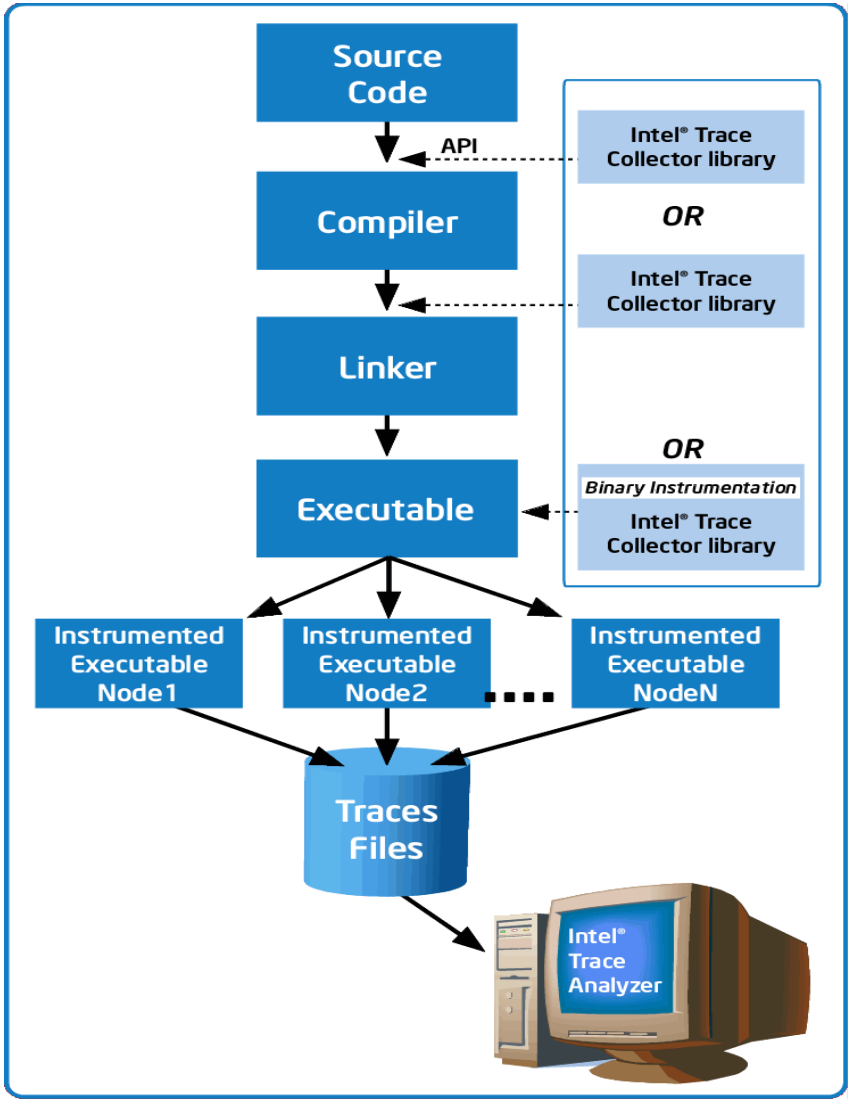
Blue = computation
Red = communication

Chart showing how the MPI processes interact

Intel® Trace Analyzer and Collector Overview

- **Intel® Trace Analyzer and Collector helps the developer:**
  - Visualize and understand parallel application behavior
  - Evaluate profiling statistics and load balancing
  - Identify communication hotspots

- **Features**
  - Event-based approach
  - Low overhead
  - Excellent scalability
  - Comparison of multiple profiles
  - Powerful aggregation and filtering functions
  - Fail-safe MPI tracing
  - Provides API to instrument user code
  - MPI correctness checking
  - Idealizer



Key Features

- Low Overhead
- Catch all MPI events
- Powerful configuration mechanism
  - Filters, settings, features
- Automatic source-code references
- Instrumentation
  - Rich API
  - Binary instrumentation (itcpin)
  - Compiler based (-tcollect)
- Fail-safe version
- Comparison of multiple profiles
- Idealizer
- MPI Correctness Checking

ITAC Prerequisites

- Set ITAC environment (per user)

```
# source /opt/intel/itac/8.1.2.033/intel64/bin/itacvars.sh impi4
```

  – Identical for host and the coprocessor

- No NFS: Upload ITAC library manually

```
# sudo scp /opt/intel/itac/8.1.2.033/mic/slib/libVT.so host-mic0:/
  lib64/
```

ITAC Usage with Intel® Xeon Phi™ Coprocessor

- Run with –trace flag (without linkage) to create a trace file
  - MPI+Offload

```
# mpirun -trace -n 2 ./test
```

  - Coprocessor only

```
# mpirun -trace -n 2 -wdir /tmp
  -host host-mic0 ./test_hello.MIC
```

  - Symmetric

```
# mpirun -trace -n 2 -host michost./test_hello :
  -wdir /tmp -n 2 -host host-mic0
  ./test_hello.MIC
```

- Flag „-trace“ will implicitly pre-load libVT.so (which finally calls libmpi.so to execute the MPI call)
- Set `VT_LOGFILE_FORMAT=stfsingle` to create a single trace

ITAC Usage with Intel® Xeon Phi™: Compilation Support

- Compile and link with „–trace“ flag

```
# mpiicc -trace -o test_hello test.c
# mpiicc –trace –mmic -o test_hello.MIC test.c
```

  - Linkage of libVT library
- Compile with –tcollect flag

```
# mpiicc –tcollect -o test_hello test.c
# mpiicc –tcollect –mmic -o test_hello.MIC test.c
```

  - Linkage of libVT library
  - Will do a full instrumentation of your code, i.e. All user functions will be visible in the trace file
  - Maximal insight, but also maximal overhead
- Use the VT API of ITAC to manually instrument your code.
- Run Intel® MPI program as usual without „-trace“ flag

```
# mpirun ...
```

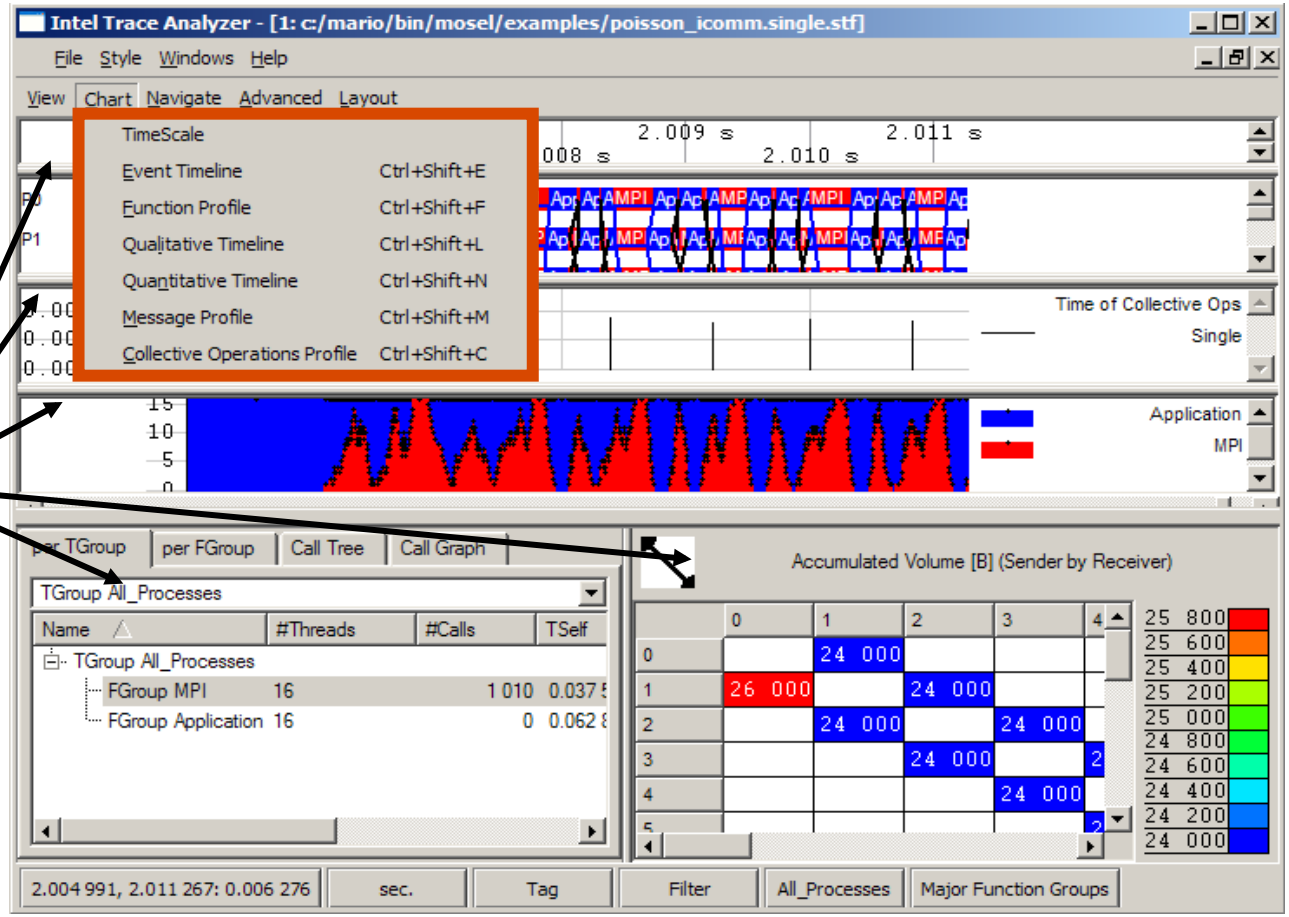
ITAC Analysis

- Start the ITAC analysis GUI with the trace file (or load it)

```
# traceanalyzer test_hello.single.stf
```

- Start the analysis, usually by inspection of the Flat Profile (default chart), the Event Timeline, and the Message Profile
  - Select “Charts->Event Timeline”
  - Select “Charts->Message Profile”
  - Zoom into the Event Timeline
    - Klick into it, keep pressed, move to the right, and release the mouse
    - See menu Navigate to get back
  - Right klick the “Group MPI->Ungroup MPI”.

Chart

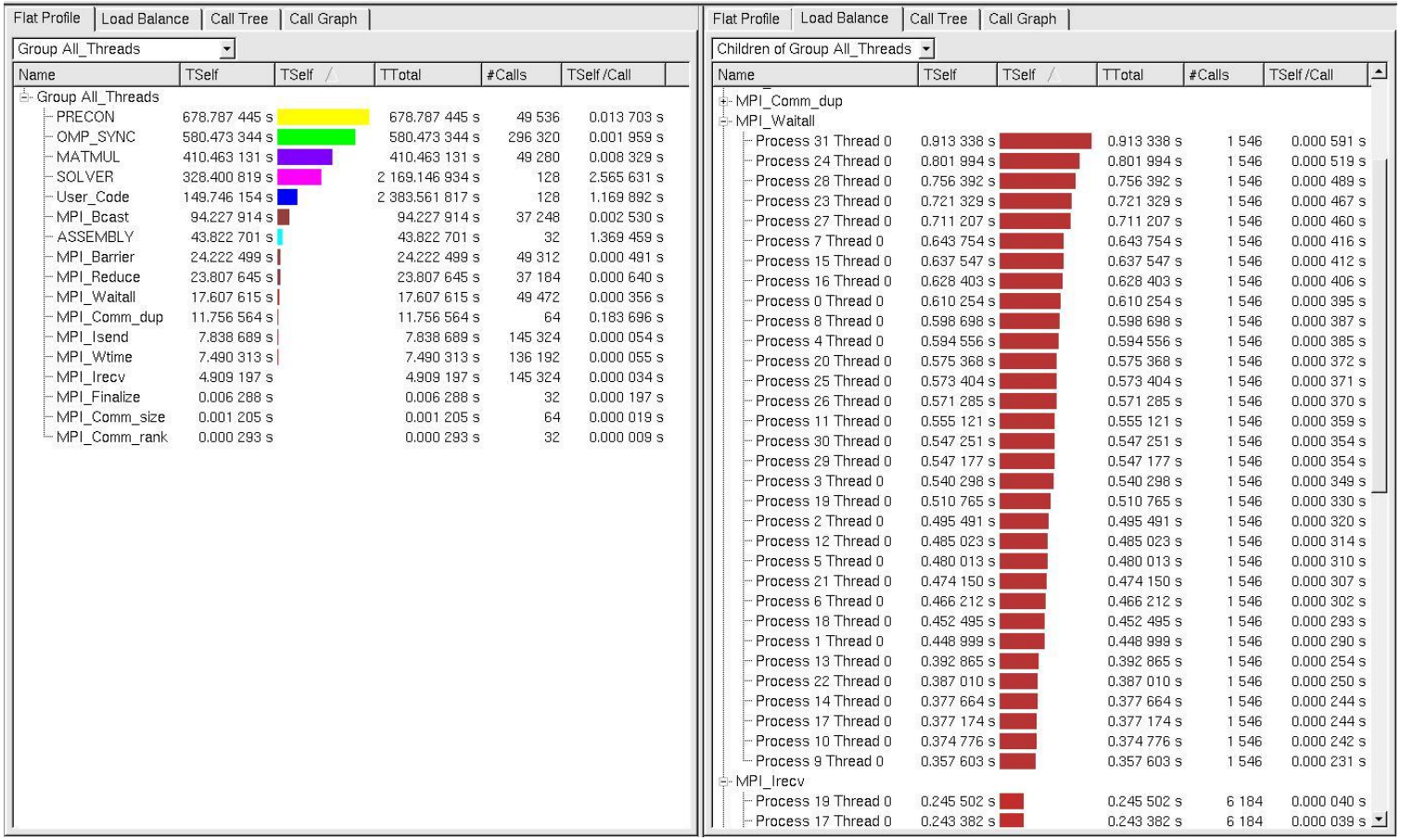


Chart

A Chart is a numerical or graphical diagram

Profiles: Flat Function Profile

- Statistics about functions



Flat Profile | Load Balance | Call Tree | Call Graph

Group All_Threads

Name	TSelf	TSelf	TTotal	#Calls	TSelf /Call
Group All_Threads					
PRECON	678.787 445 s		678.787 445 s	49 536	0.013 703 s
OMP_SYNC	580.473 344 s		580.473 344 s	296 320	0.001 959 s
MATMUL	410.463 131 s		410.463 131 s	49 280	0.008 329 s
SOLVER	328.400 819 s		2 169.146 934 s	128	2.565 631 s
User_Code	149.746 154 s		2 383.561 817 s	128	1.169 892 s
MPI_Bcast	94.227 914 s		94.227 914 s	37 248	0.002 530 s
ASSEMBLY	43.822 701 s		43.822 701 s	32	1.369 459 s
MPI_Barrier	24.222 499 s		24.222 499 s	49 312	0.000 491 s
MPI_Reduce	23.807 645 s		23.807 645 s	37 184	0.000 640 s
MPI_Waitall	17.607 615 s		17.607 615 s	49 472	0.000 356 s
MPI_Comm_dup	11.756 564 s		11.756 564 s	64	0.183 696 s
MPI_Isend	7.838 689 s		7.838 689 s	145 324	0.000 054 s
MPI_Wtime	7.490 313 s		7.490 313 s	136 192	0.000 055 s
MPI_Irecv	4.909 197 s		4.909 197 s	145 324	0.000 034 s
MPI_Finalize	0.006 288 s		0.006 288 s	32	0.000 197 s
MPI_Comm_size	0.001 205 s		0.001 205 s	64	0.000 019 s
MPI_Comm_rank	0.000 293 s		0.000 293 s	32	0.000 009 s

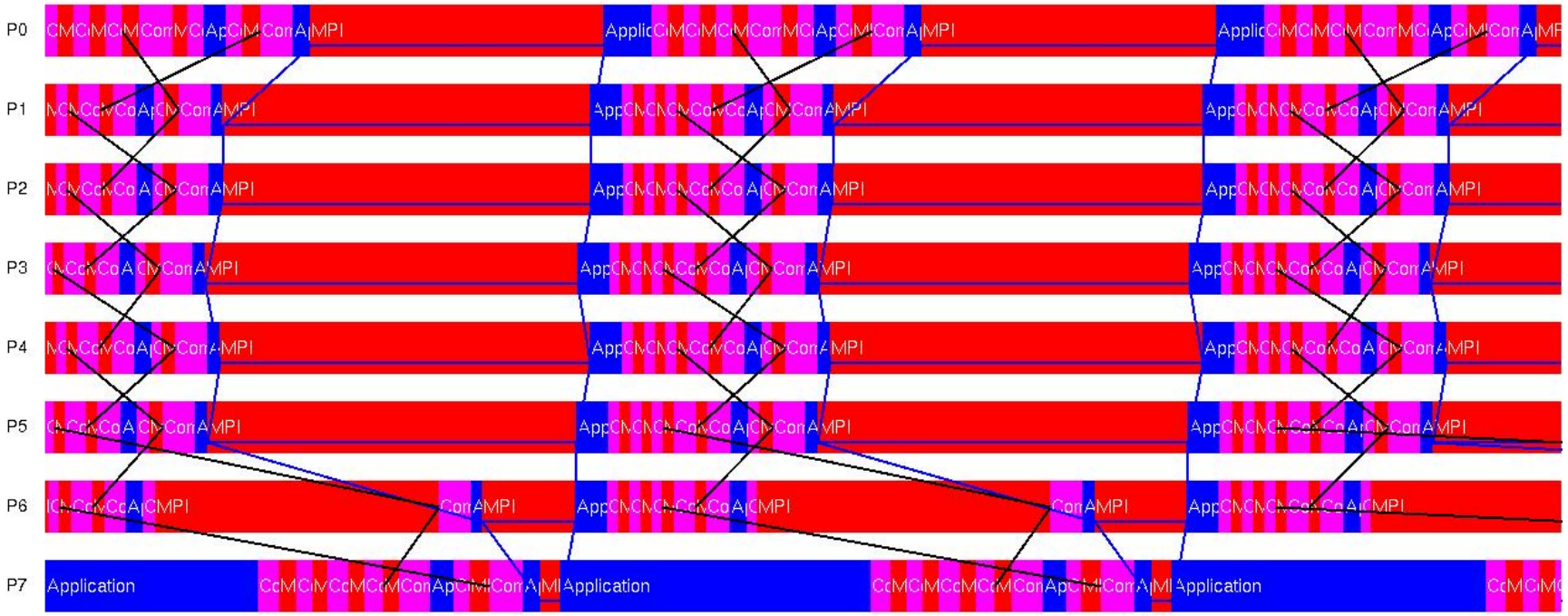
Flat Profile | Load Balance | Call Tree | Call Graph

Children of Group All_Threads

Name	TSelf	TSelf	TTotal	#Calls	TSelf /Call
MPI_Comm_dup					
MPI_Waitall					
Process 31 Thread 0	0.913 338 s		0.913 338 s	1 546	0.000 591 s
Process 24 Thread 0	0.801 994 s		0.801 994 s	1 546	0.000 519 s
Process 28 Thread 0	0.756 392 s		0.756 392 s	1 546	0.000 489 s
Process 23 Thread 0	0.721 329 s		0.721 329 s	1 546	0.000 467 s
Process 27 Thread 0	0.711 207 s		0.711 207 s	1 546	0.000 460 s
Process 7 Thread 0	0.643 754 s		0.643 754 s	1 546	0.000 416 s
Process 15 Thread 0	0.637 547 s		0.637 547 s	1 546	0.000 412 s
Process 16 Thread 0	0.628 403 s		0.628 403 s	1 546	0.000 406 s
Process 0 Thread 0	0.610 254 s		0.610 254 s	1 546	0.000 395 s
Process 8 Thread 0	0.598 698 s		0.598 698 s	1 546	0.000 387 s
Process 4 Thread 0	0.594 556 s		0.594 556 s	1 546	0.000 385 s
Process 20 Thread 0	0.575 368 s		0.575 368 s	1 546	0.000 372 s
Process 25 Thread 0	0.573 404 s		0.573 404 s	1 546	0.000 371 s
Process 26 Thread 0	0.571 285 s		0.571 285 s	1 546	0.000 370 s
Process 11 Thread 0	0.555 121 s		0.555 121 s	1 546	0.000 359 s
Process 30 Thread 0	0.547 251 s		0.547 251 s	1 546	0.000 354 s
Process 29 Thread 0	0.547 177 s		0.547 177 s	1 546	0.000 354 s
Process 3 Thread 0	0.540 298 s		0.540 298 s	1 546	0.000 349 s
Process 19 Thread 0	0.510 765 s		0.510 765 s	1 546	0.000 330 s
Process 2 Thread 0	0.495 491 s		0.495 491 s	1 546	0.000 320 s
Process 12 Thread 0	0.485 023 s		0.485 023 s	1 546	0.000 314 s
Process 5 Thread 0	0.480 013 s		0.480 013 s	1 546	0.000 310 s
Process 21 Thread 0	0.474 150 s		0.474 150 s	1 546	0.000 307 s
Process 6 Thread 0	0.466 212 s		0.466 212 s	1 546	0.000 302 s
Process 18 Thread 0	0.452 495 s		0.452 495 s	1 546	0.000 293 s
Process 1 Thread 0	0.448 999 s		0.448 999 s	1 546	0.000 290 s
Process 13 Thread 0	0.392 865 s		0.392 865 s	1 546	0.000 254 s
Process 22 Thread 0	0.387 010 s		0.387 010 s	1 546	0.000 250 s
Process 14 Thread 0	0.377 664 s		0.377 664 s	1 546	0.000 244 s
Process 17 Thread 0	0.377 174 s		0.377 174 s	1 546	0.000 244 s
Process 10 Thread 0	0.374 776 s		0.374 776 s	1 546	0.000 242 s
Process 9 Thread 0	0.357 603 s		0.357 603 s	1 546	0.000 231 s
MPI_Irecv					
Process 19 Thread 0	0.245 502 s		0.245 502 s	6 184	0.000 040 s
Process 17 Thread 0	0.243 382 s		0.243 382 s	6 184	0.000 039 s

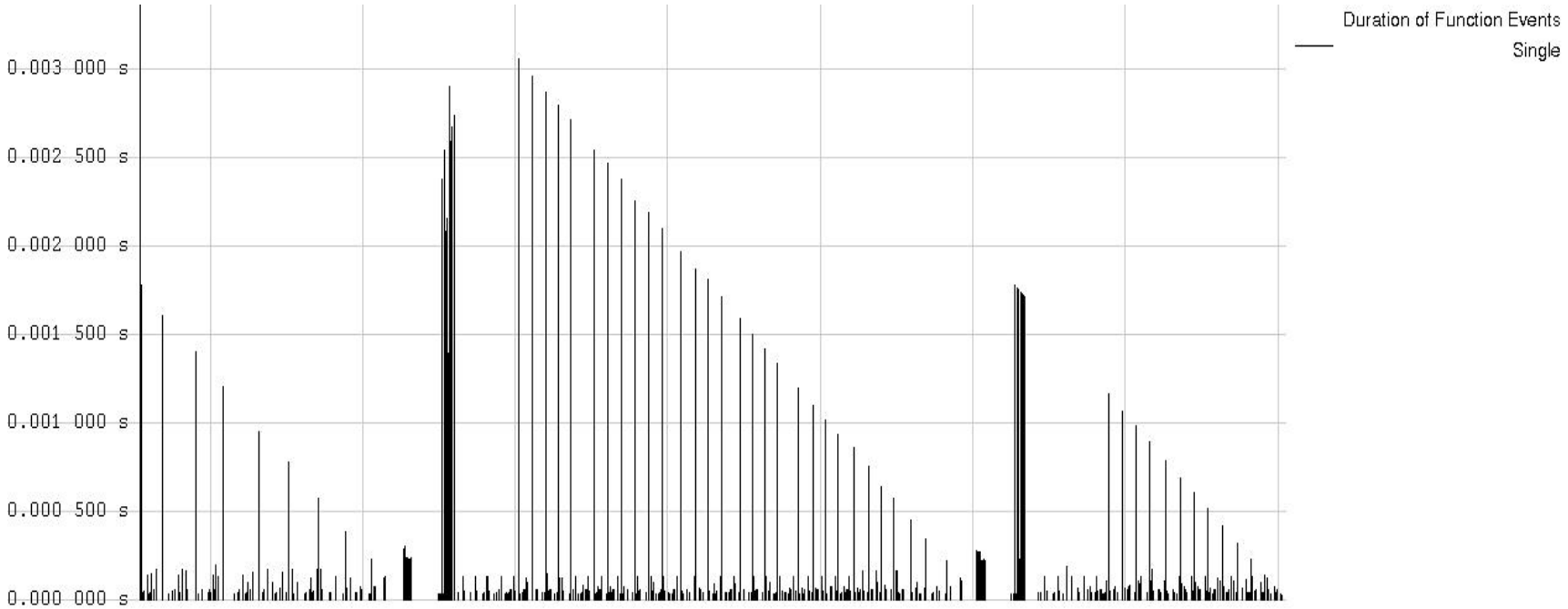
Timelines: Event Timeline

- Get impression of program structure
- Display functions, messages and collective operations for each process/thread along time-axis
- Retrieval of detailed event information



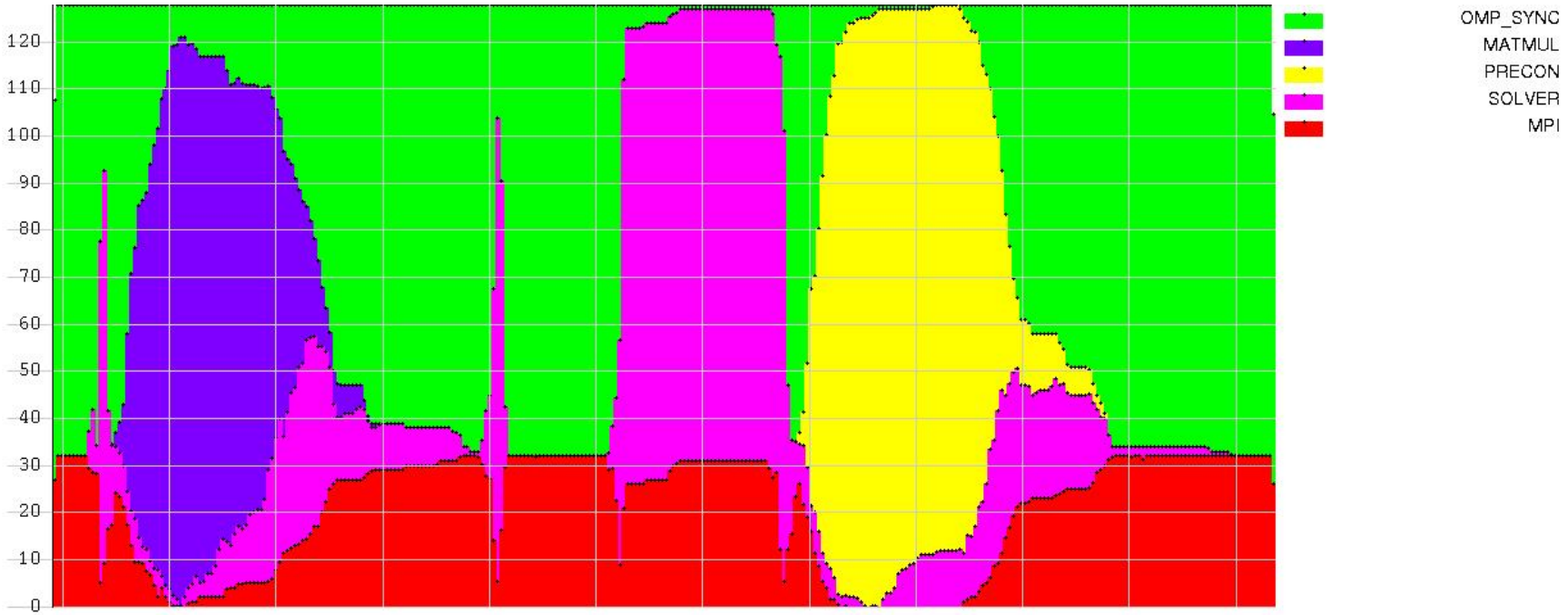
Timelines: Qualitative Timeline

- Find patterns and irregularities
- Display attributes of functions, messages or collective operations as they occur for any process/thread
- Retrieval of detailed event information



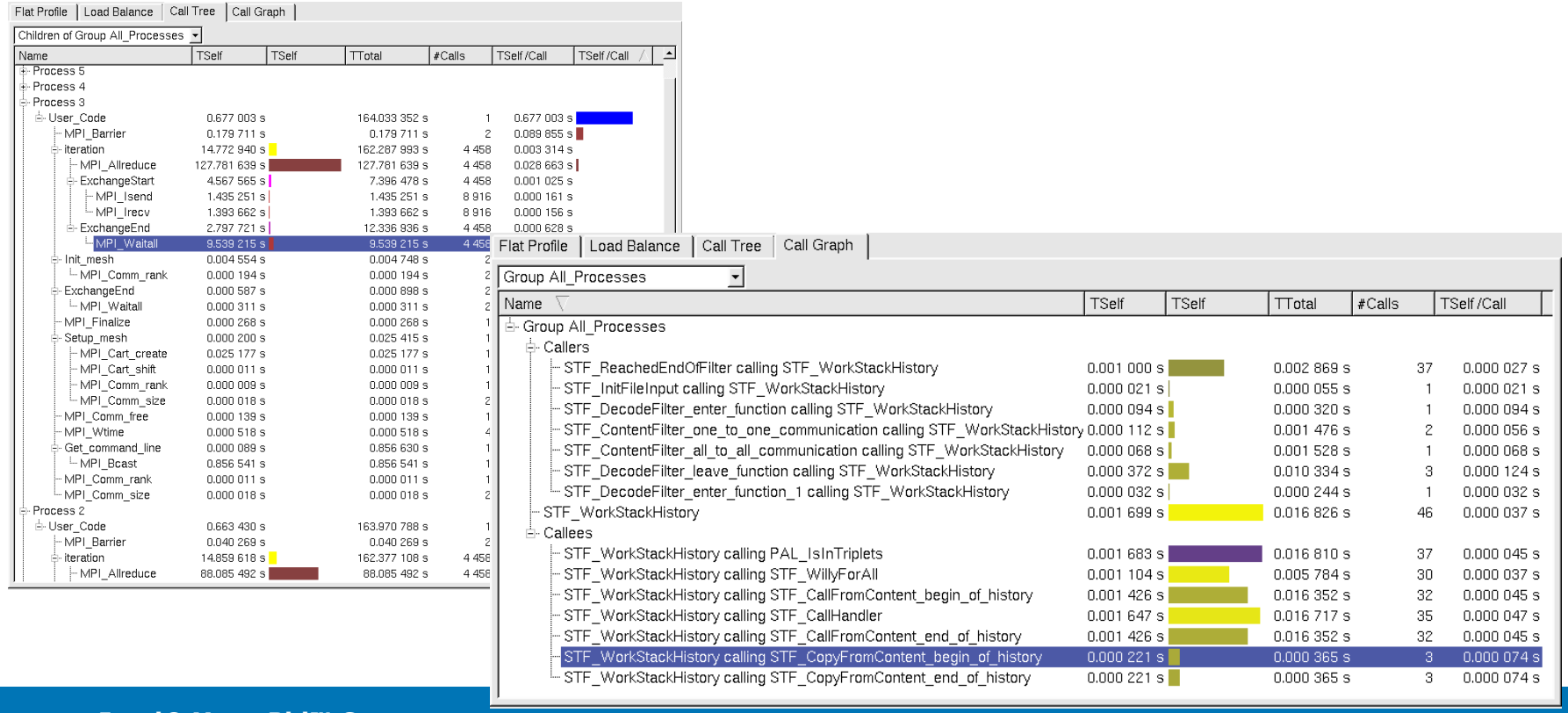
Timelines: Quantitative Timeline

- Get impression on parallelism and load balance
- Show for every function how many threads/processes are currently executing it



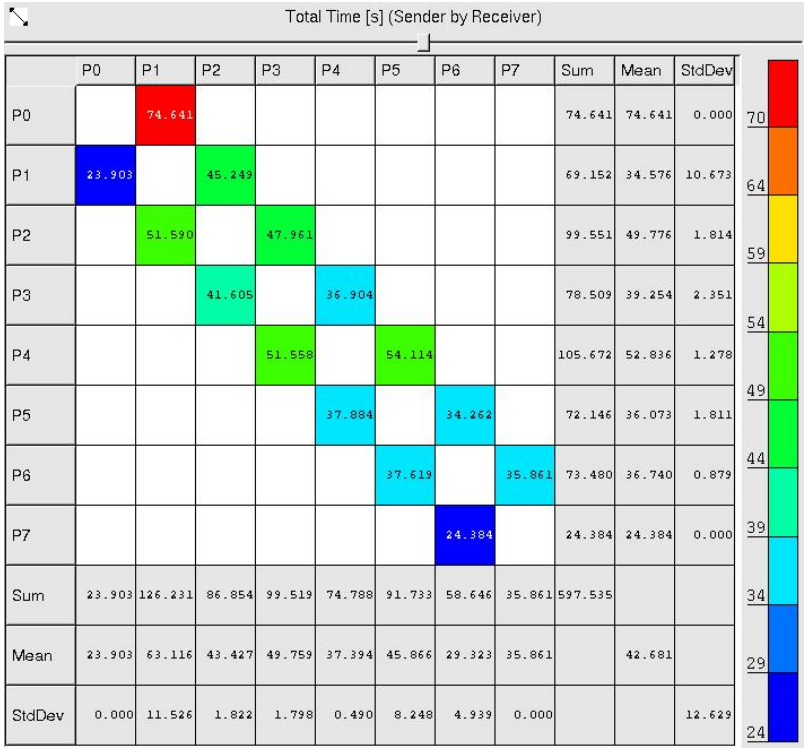
Profiles: Call-Tree and Call-Graph

- Function statistics including calling hierarchy
  - Tree: call-stack
  - Graph: calling dependencies

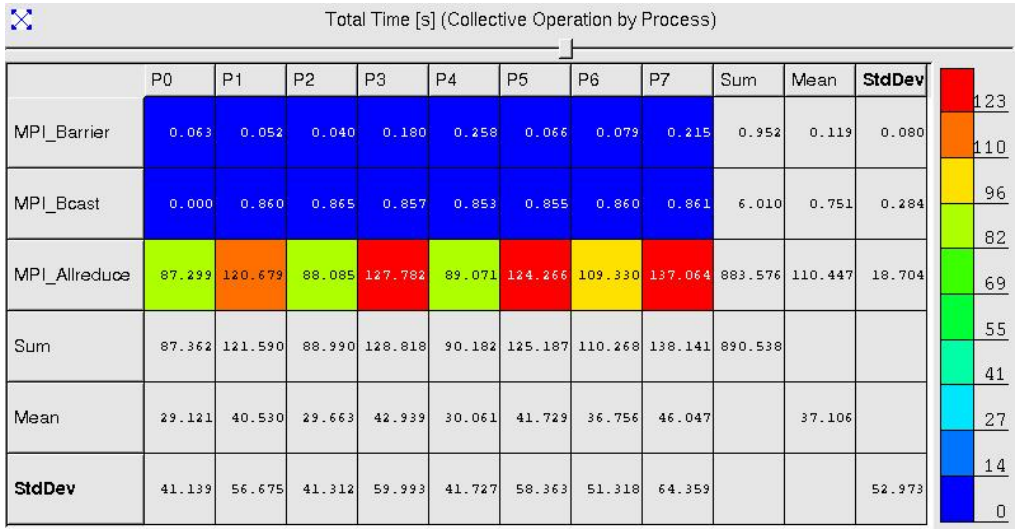


Communication Profiles

- Statistics about point-to-point or collective communication
- Generic matrix supports grouping by several attributes in each dimension
  Sender, Receiver, Data volume per msg, Tag, Communicator, Type
- Available attributes: Count, Bytes transferred, Time, Transfer rate



	P0	P1	P2	P3	P4	P5	P6	P7	Sum	Mean	StdDev
P0		74.641							74.641	74.641	0.000
P1	23.903		45.249						69.152	34.576	10.673
P2		51.590		47.961					99.551	49.776	1.814
P3			41.605		36.904				78.509	39.254	2.351
P4				51.558		54.114			105.672	52.836	1.278
P5					37.884		34.262		72.146	36.073	1.811
P6						37.619		35.861	73.480	36.740	0.879
P7							24.384		24.384	24.384	0.000
Sum	23.903	126.231	86.854	99.519	74.788	91.733	58.646	35.861	597.535		
Mean	23.903	63.116	43.427	49.759	37.394	45.866	29.323	35.861		42.681	
StdDev	0.000	11.526	1.822	1.798	0.490	8.248	4.939	0.000			12.629

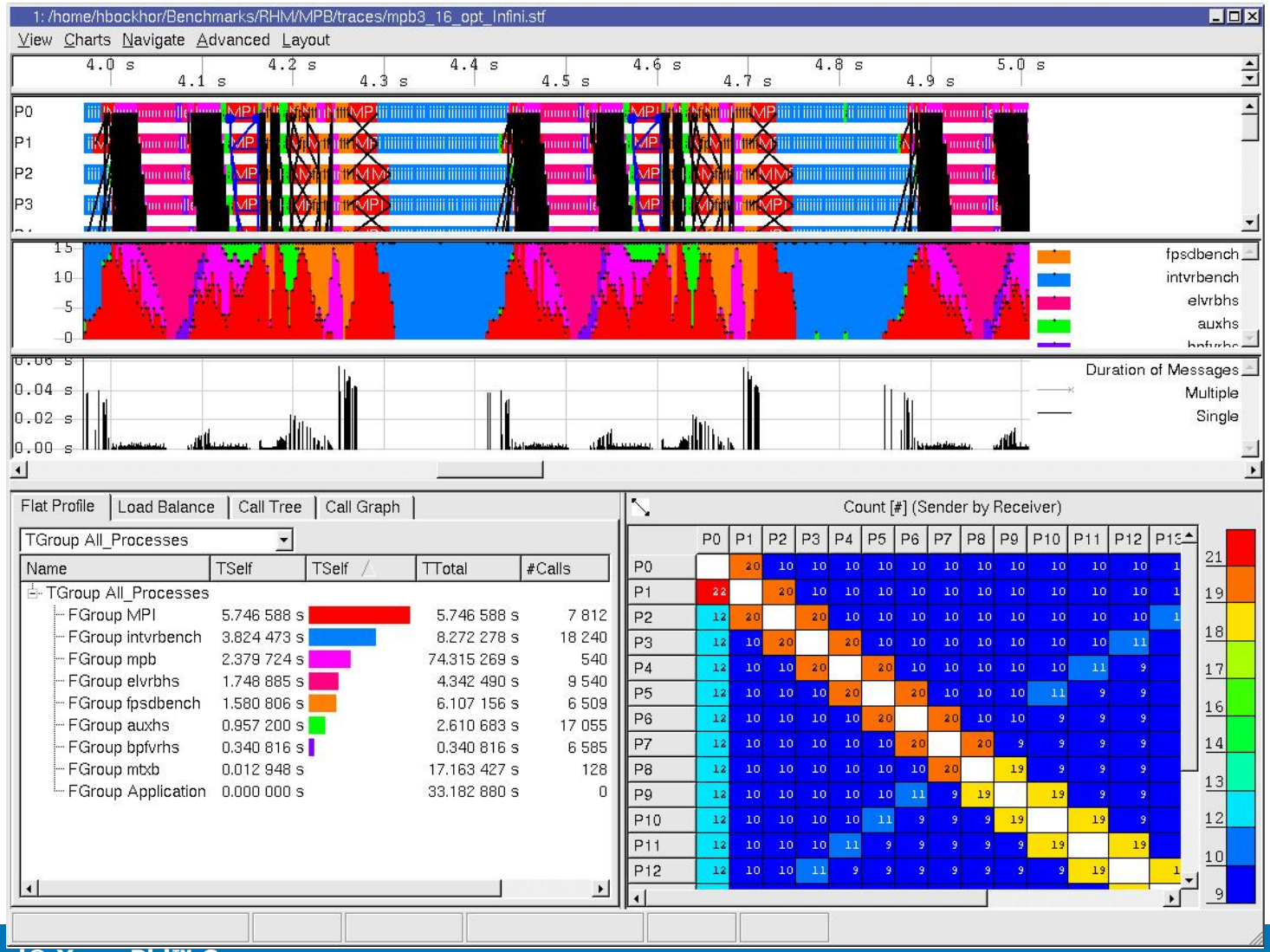


	P0	P1	P2	P3	P4	P5	P6	P7	Sum	Mean	StdDev
MPI_Barrier	0.063	0.052	0.040	0.180	0.258	0.066	0.079	0.215	0.952	0.119	0.080
MPI_Bcast	0.000	0.860	0.865	0.857	0.853	0.855	0.860	0.861	6.010	0.751	0.284
MPI_Allreduce	87.299	120.679	88.085	127.782	89.071	124.266	109.330	137.064	883.576	110.447	18.704
Sum	87.362	121.590	88.990	128.818	90.182	125.187	110.268	138.141	890.538		
Mean	29.121	40.530	29.663	42.939	30.061	41.729	36.756	46.047		37.106	
StdDev	41.139	56.675	41.312	59.993	41.727	58.363	51.318	64.359			52.973

View

- Helps navigating through the trace data and keep orientation
- Every View can contain several Charts
- A View on a file is defined by a triplet of
  - time-span
  - set of threads
  - set of functions
- All Charts follow changes to View (e.g. zooming)
- Timelines are correctly aligned along time

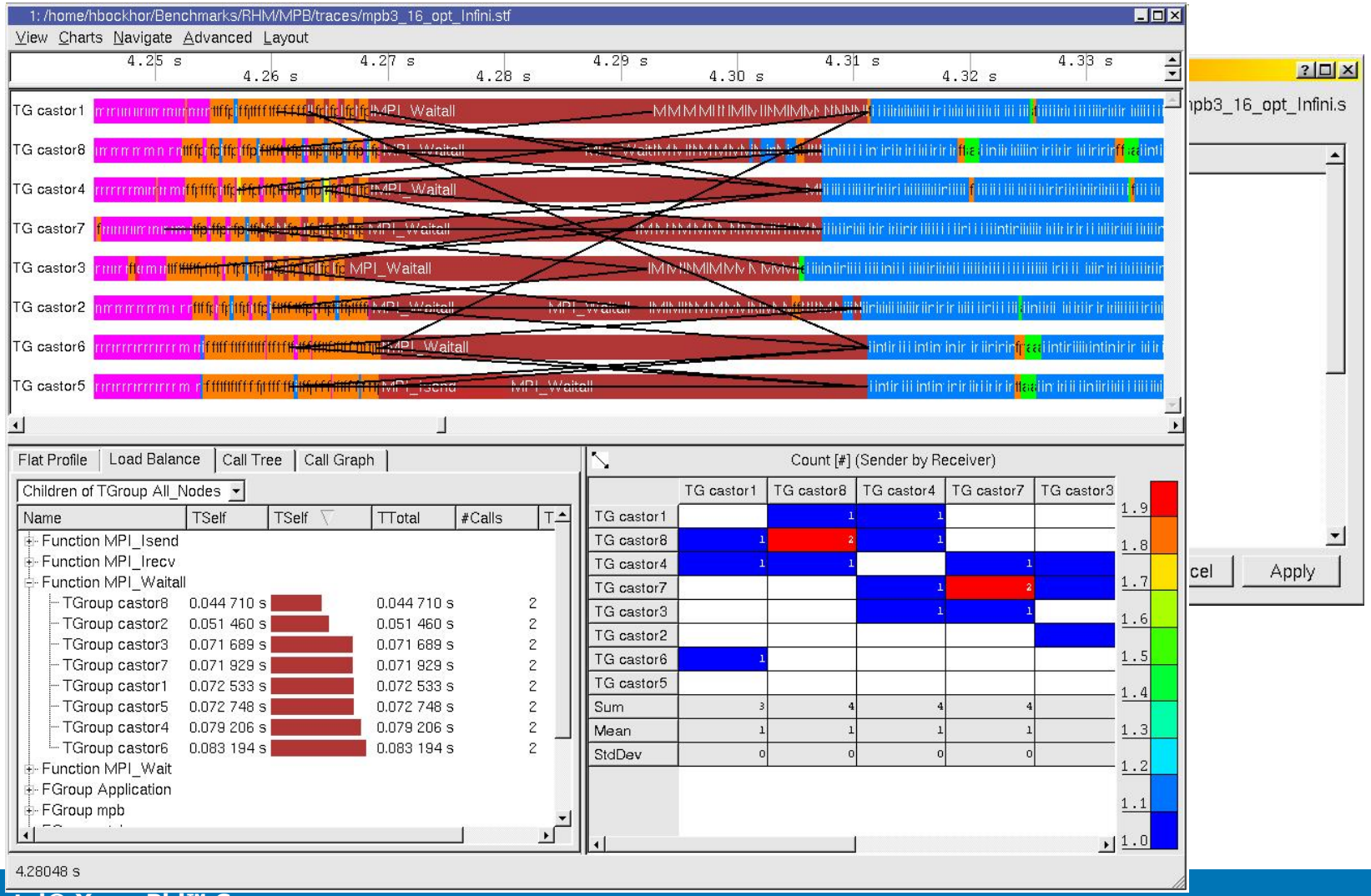
View - zooming



Grouping and Aggregation

- Allow analysis on different levels of detail by aggregating data upon group-definitions
- Functions and threads can be grouped hierarchically
  - Function Groups and Thread Groups
- Arbitrary nesting is supported
  - Functions/threads on the same level as groups
  - User can define his/her own groups
- Aggregation is part of View-definition
  - All charts in a View adapt to requested grouping
  - All charts support aggregation

Aggregation Example



Tagging & Filtering

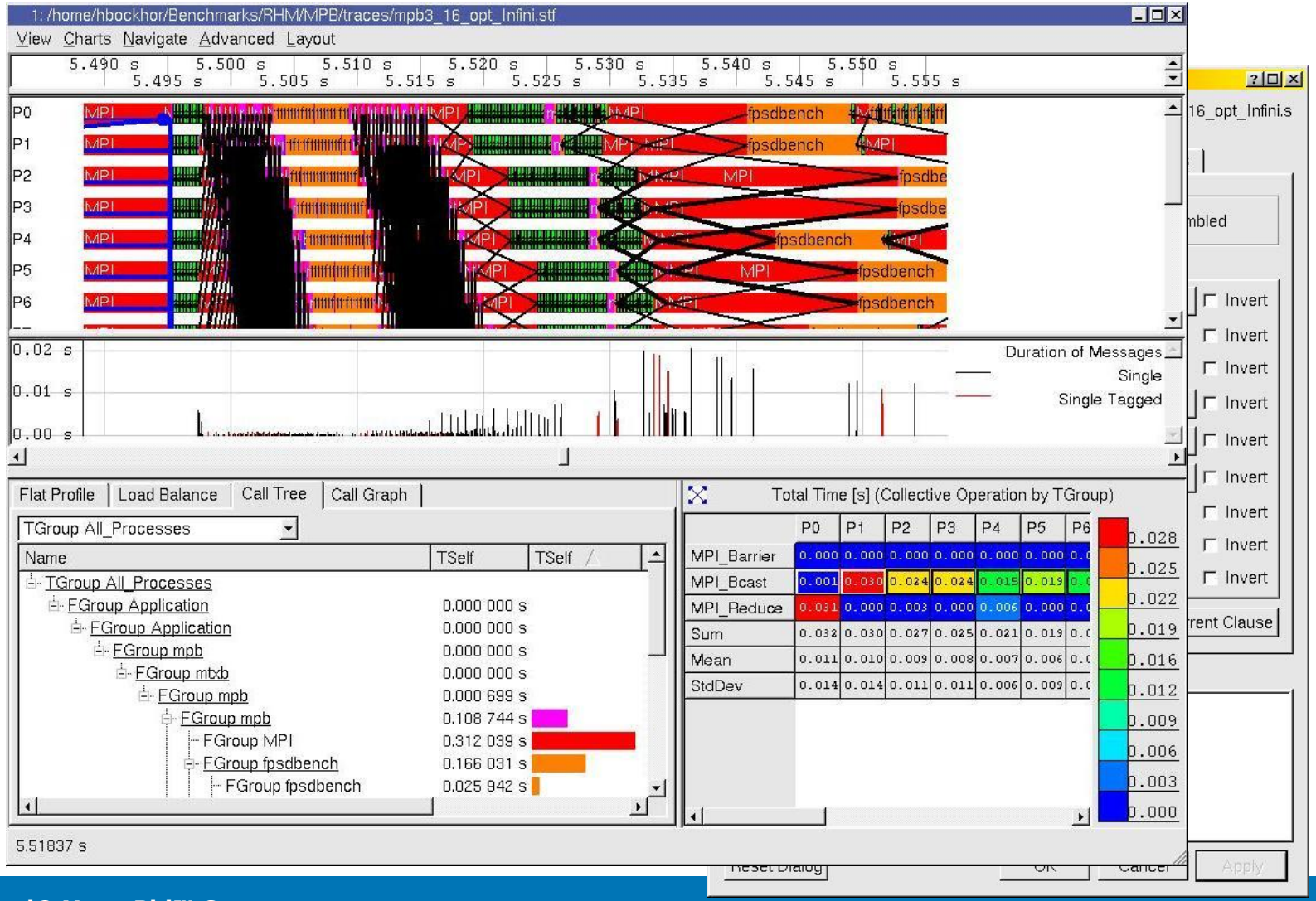
- Help concentrating on relevant parts
- Avoid getting lost in huge amounts of trace data

Define a set of interesting data
  - E.g. all occurrences of function x
  - E.g. all messages with tag y on communicator z

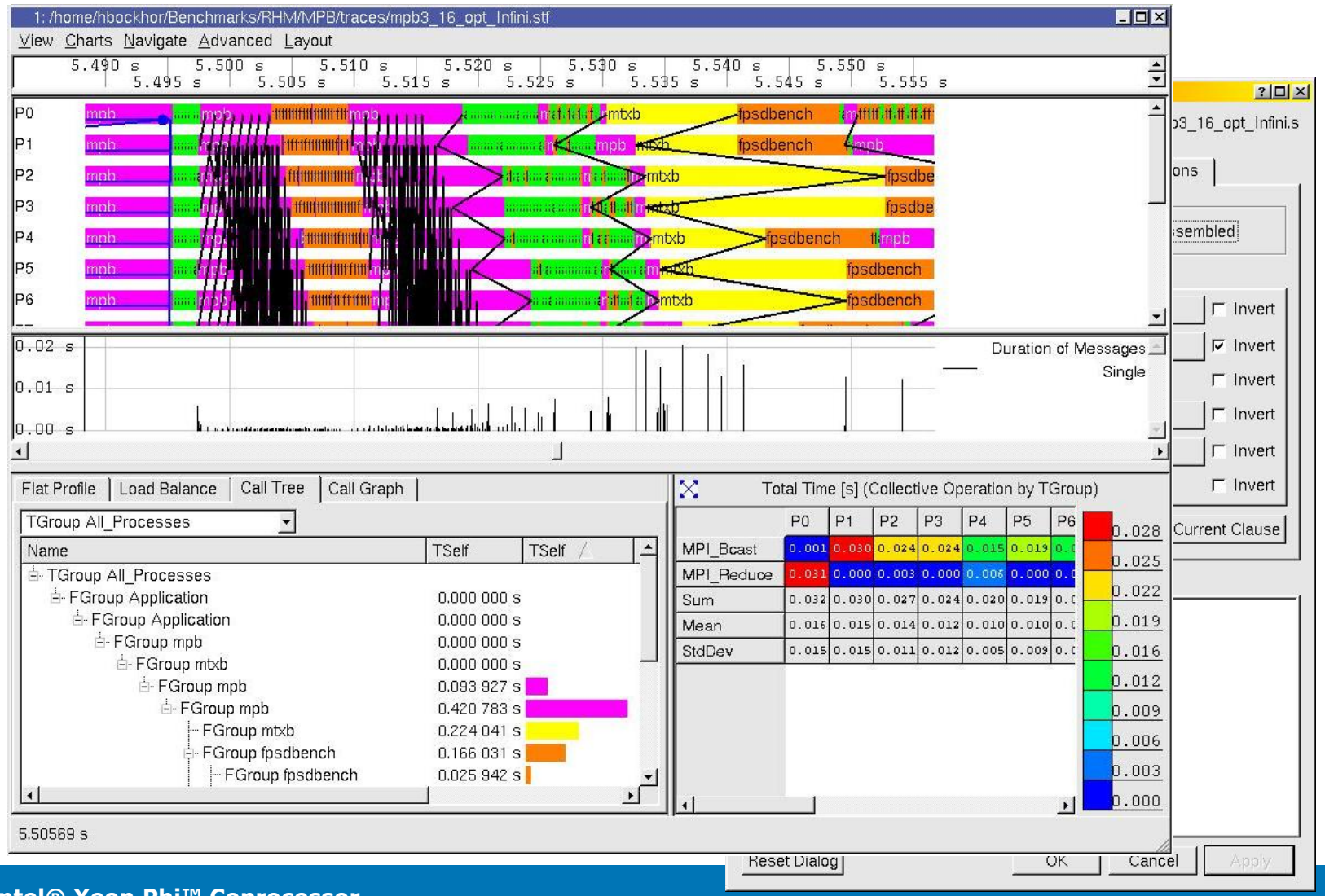
Combine several filters:
  Intersection, Union, Complement

Apply it
  - Tagging: Highlight messages
  - Filtering: Suppress all non-matching events

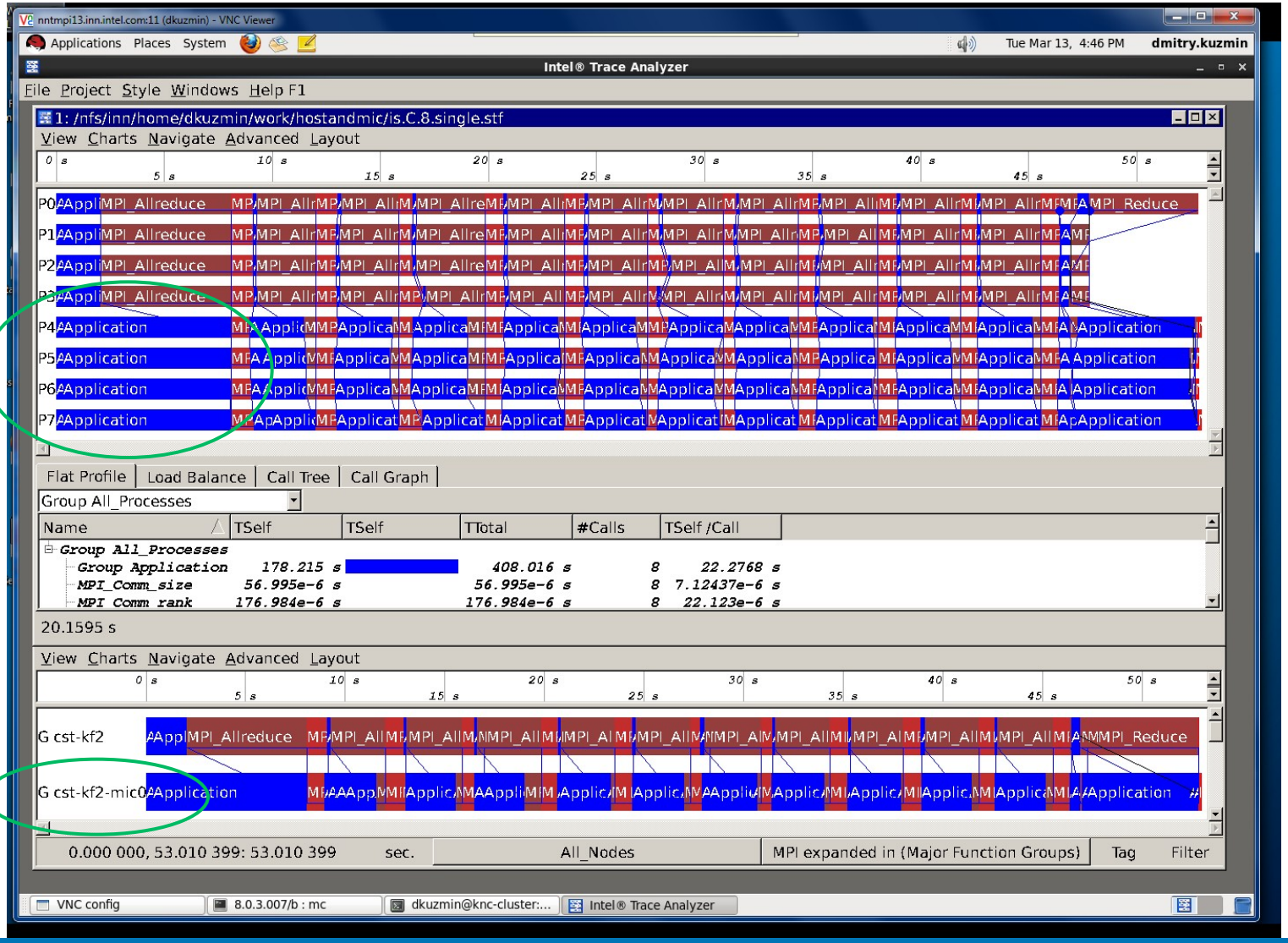
Tagging Example



Filtering Example



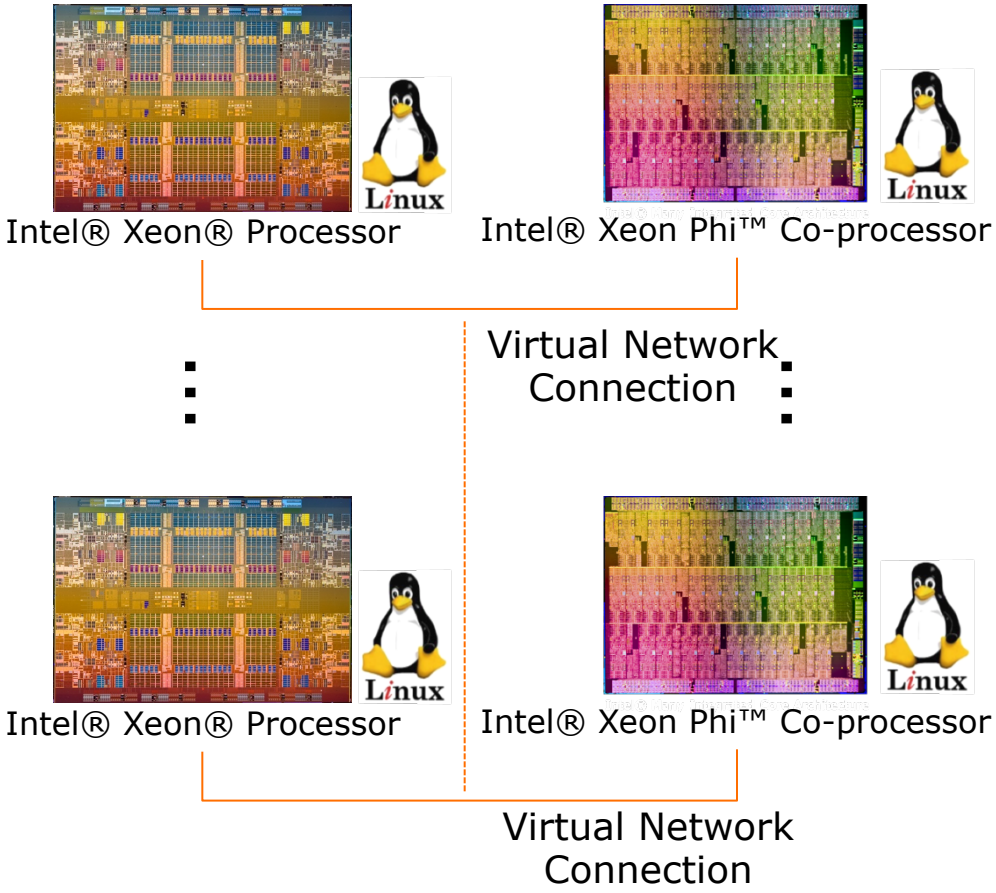
Full ITAC Functionality on Intel® Xeon Phi™



Outline

- Overview
- Prerequisites of Intel® MPI
- Programming Models
- Hybrid Computing
- Intel® Trace Analyzer and Collector
- Load Balancing
- Debugging

Intel® Xeon Phi™ Coprocessor Becomes a Network Node



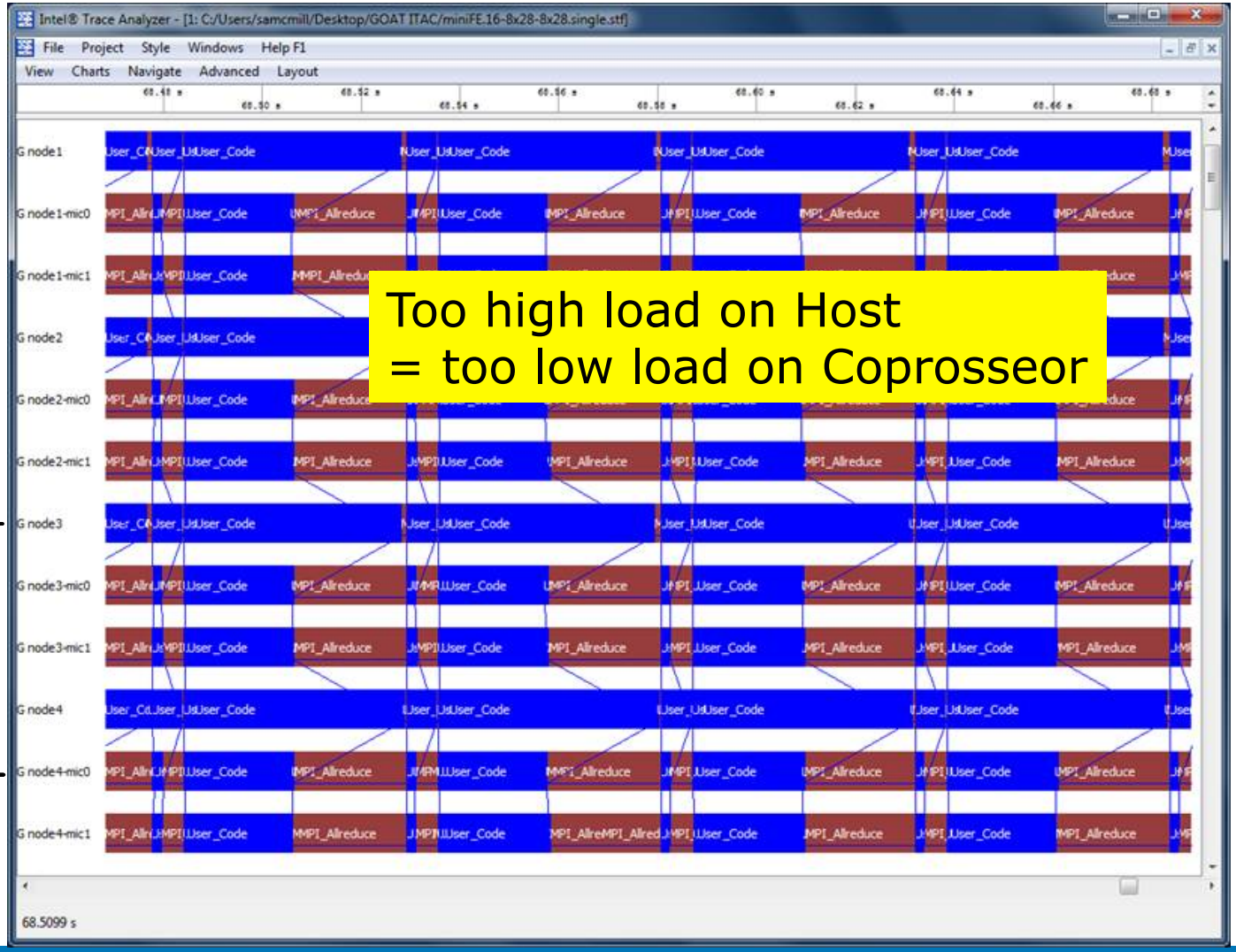
Intel® Intel® Xeon Phi™ Architecture + Linux enables IP addressability

Load Balancing

- Situation
  - Host and coprocessor computation performance are different
  - Host and coprocessor internal communication speed is different
- MPI in symmetric mode is like running on a heterogenous cluster
- Load balanced codes (on homogeneous cluser) may get imbalanced!
- Solution? No general solution!
  - Approach 1: Adapt MPI mapping of (hybrid) code to performance characteristics: #m processes per host, #n processes per coprocessor(s)
  - Approach 2: Change code internal mapping of workload to MPI processes
    - Example: uneven split of calculation grid for MPI processes on host vs. coprocessors(s)
  - Approach 3: ...
- Pre-work: Analyze load balance of application with ITAC
  - Ideal Interconnect Simulator

Improving Load Balance: Real World Case

Collapsed data per node and coprocessor



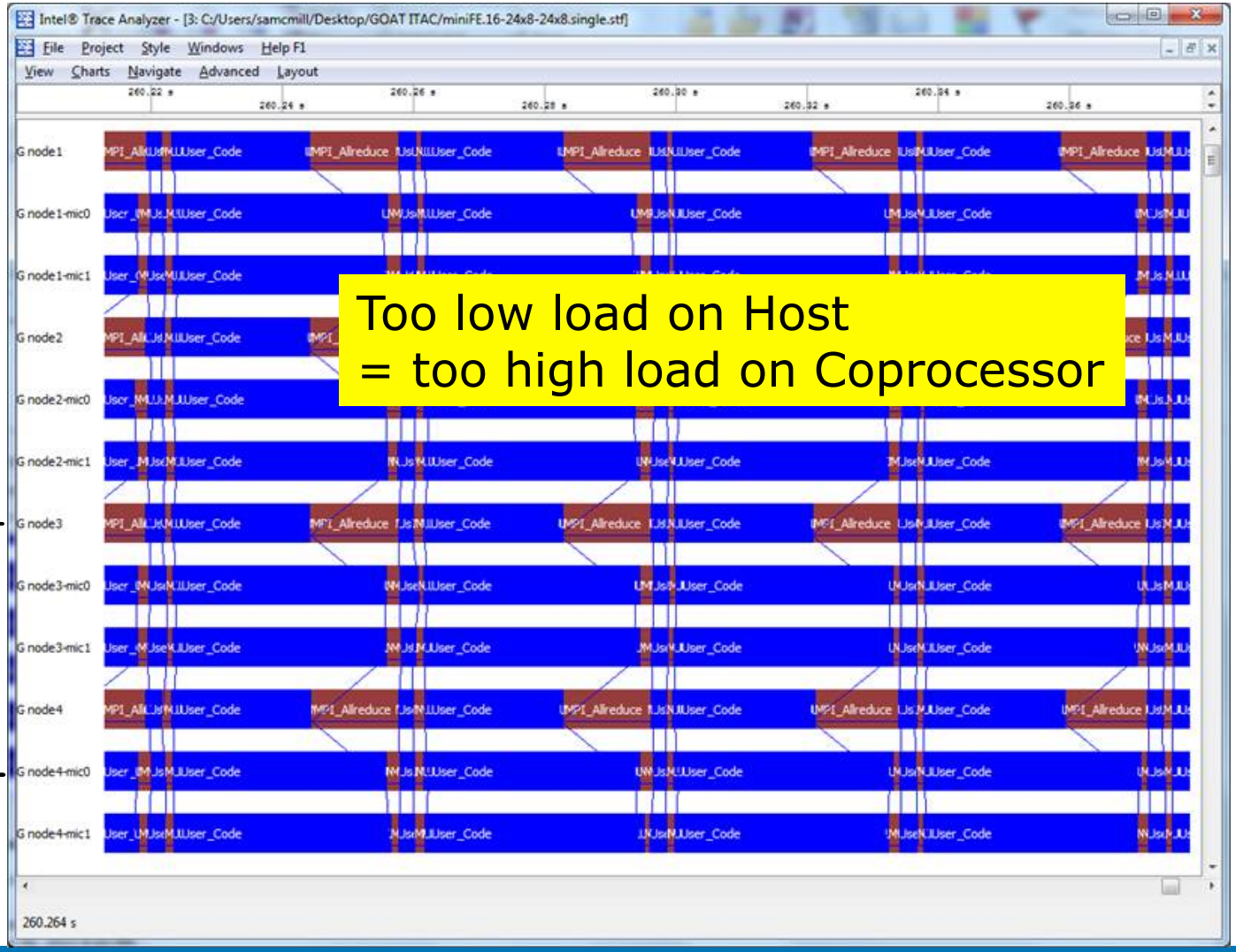
Too high load on Host
= too low load on Coprosseor

Host
16 MPI procs x
1 OpenMP thread

Coprocessor
8 MPI procs x
28 OpenMP threads

Improving Load Balance: Real World Case

Collapsed data per node and coprocessor

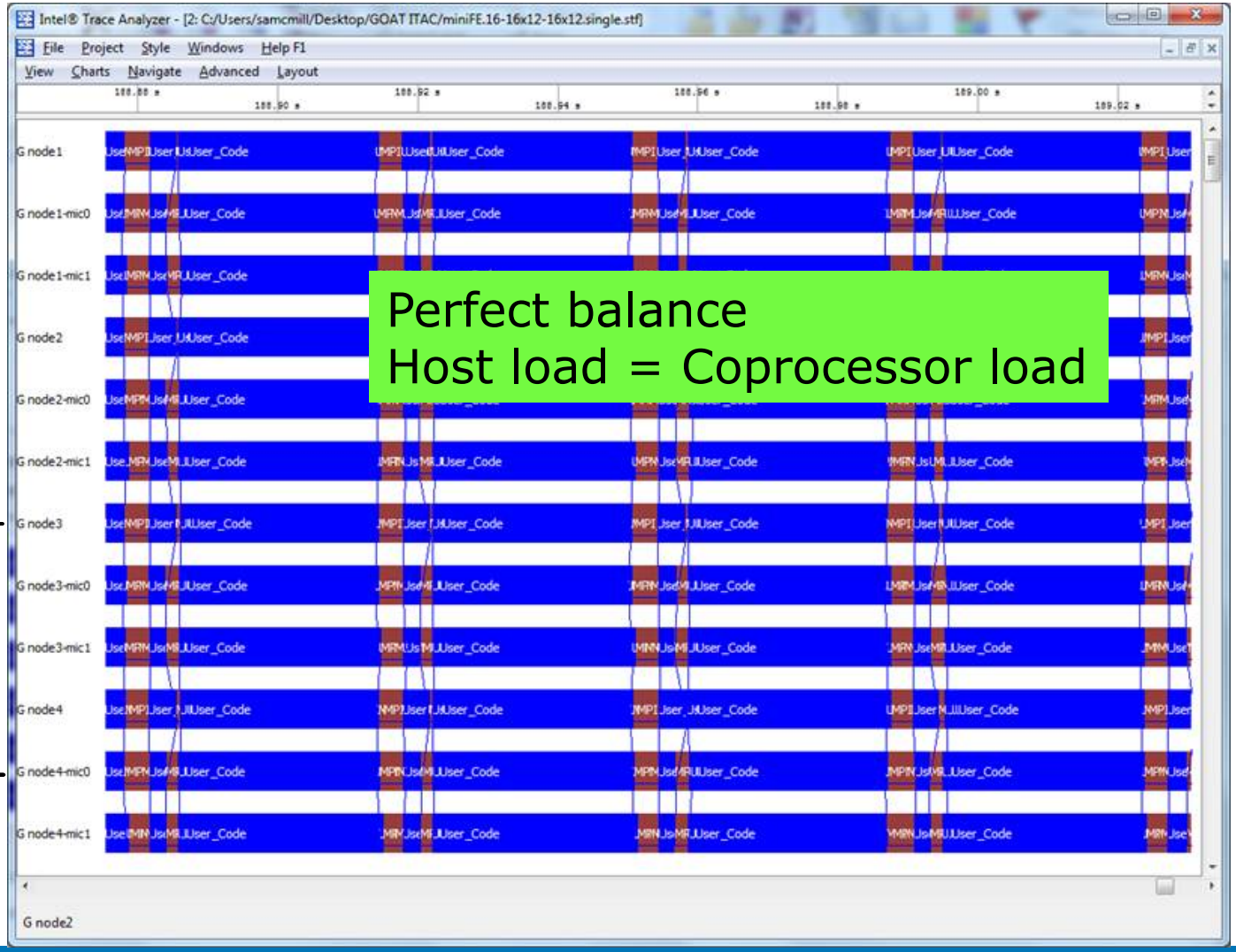


Host
16 MPI procs x
1 OpenMP thread

Coprocessor
24 MPI procs x
8 OpenMP threads

Improving Load Balance: Real World Case

Collapsed data per node and coprocessor



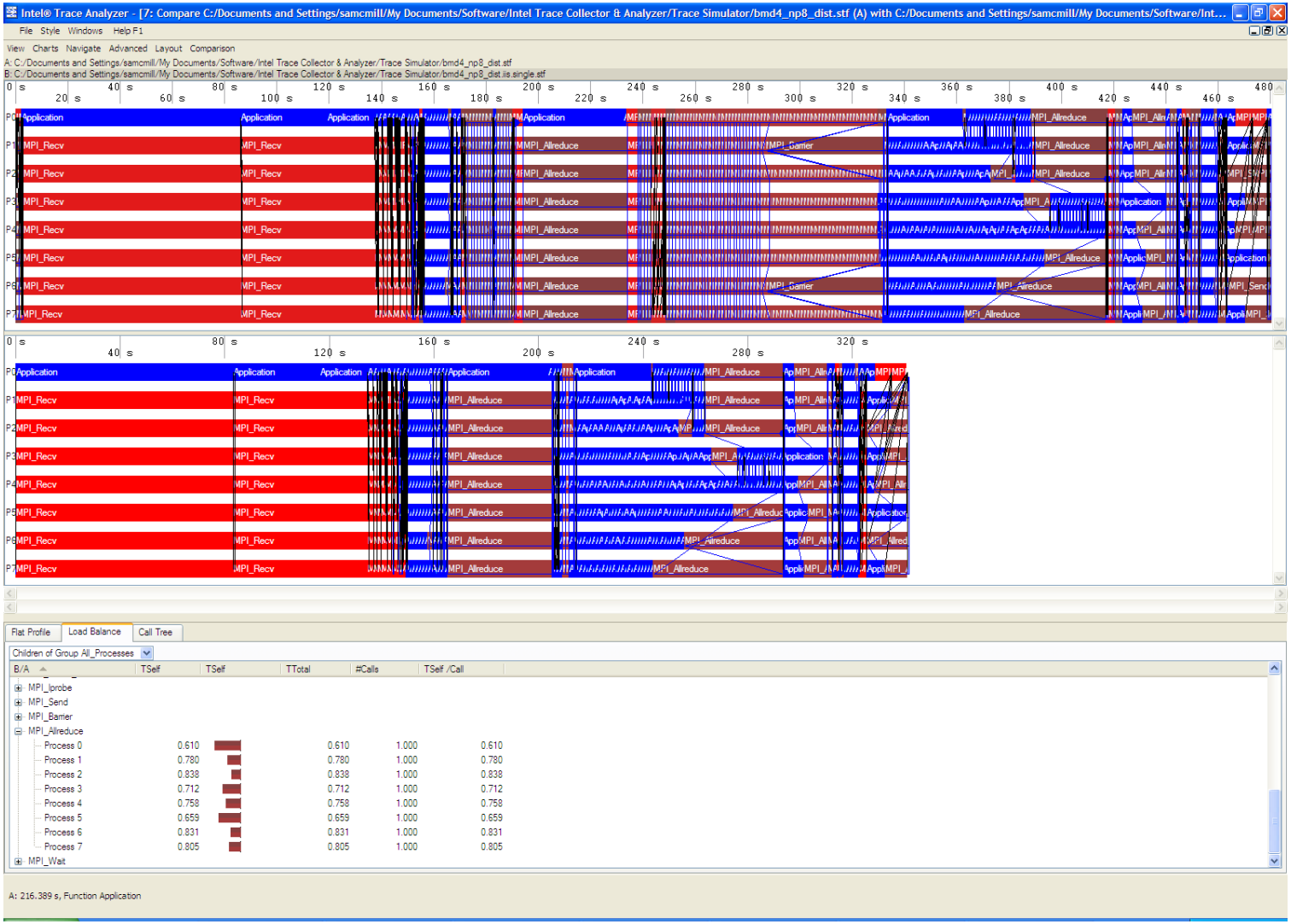
Host
16 MPI procs x
1 OpenMP thread

Coprocessor
16 MPI procs x
12 OpenMP thrds

Ideal Interconnect Simulator (Idealizer)

- What is the Ideal Interconnect Simulator?
  - Using a ITAC trace of an MPI application, simulate it under ideal conditions
    - Zero network latency
    - Infinite network bandwidth
    - Zero MPI buffer copy time
    - Infinite MPI buffer size
  - Only limiting factors are concurrency rules, e.g.,
    - A message can not be received before it is sent
    - An All-to-All collective may end only when the last thread starts

Ideal Interconnect Simulator (Idealizer)

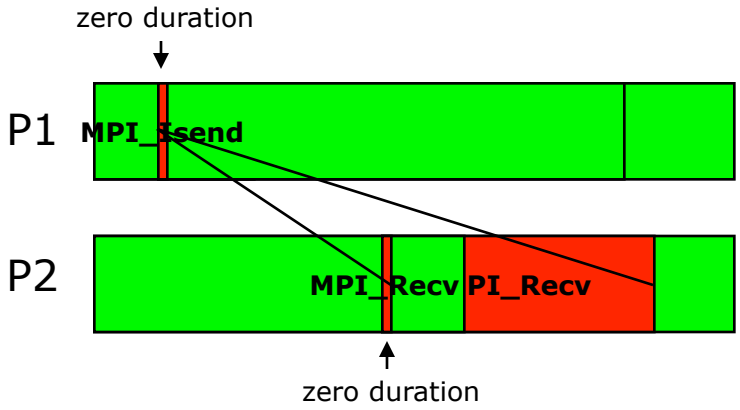


Actual trace

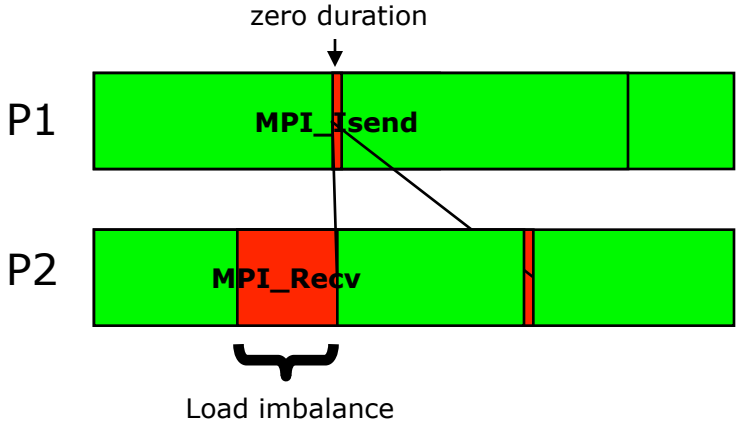
Idealized Trace

Building Blocks: Elementary Messages

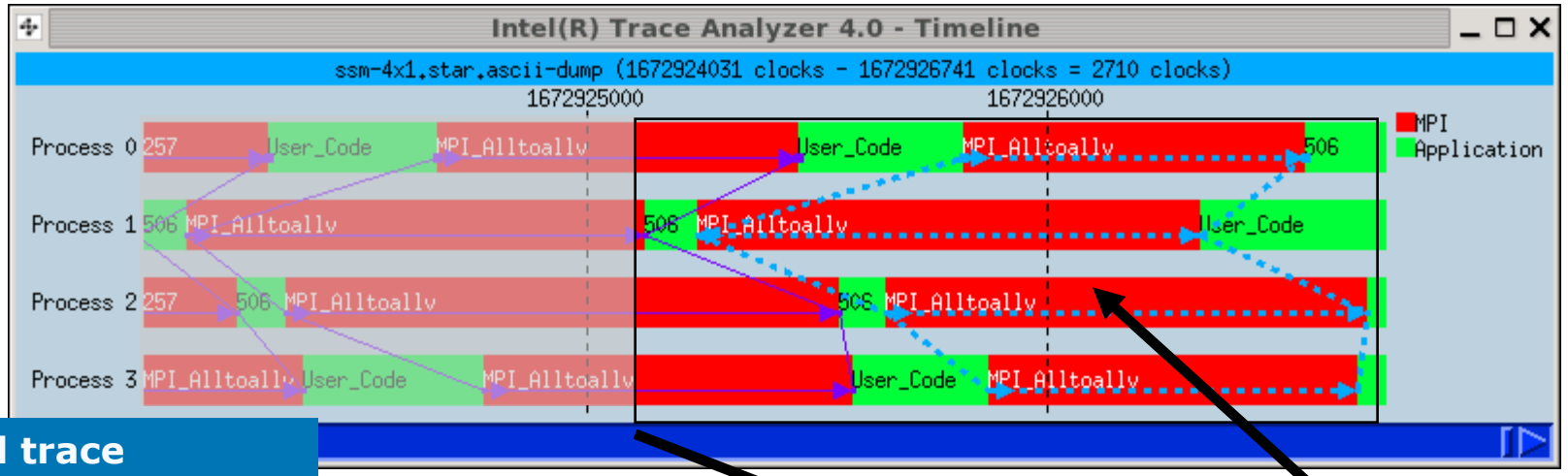
Early Send / Late Receive



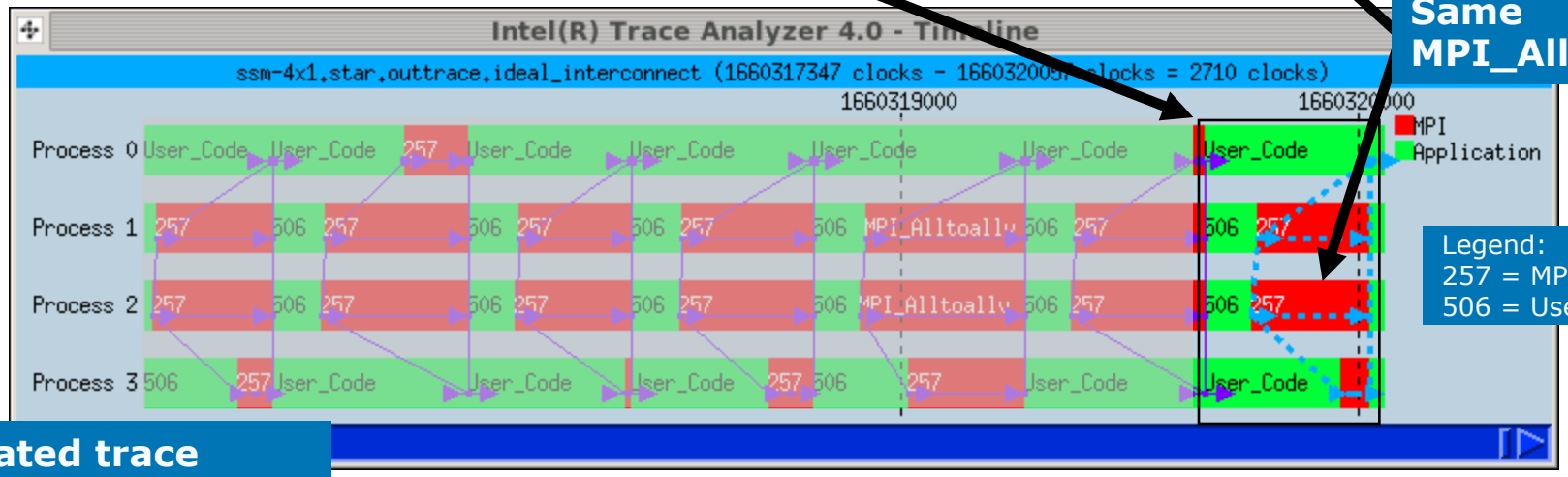
Late Send / Early Receive



Building Blocks: Collective Operations



Actual trace (Gigabit Ethernet)



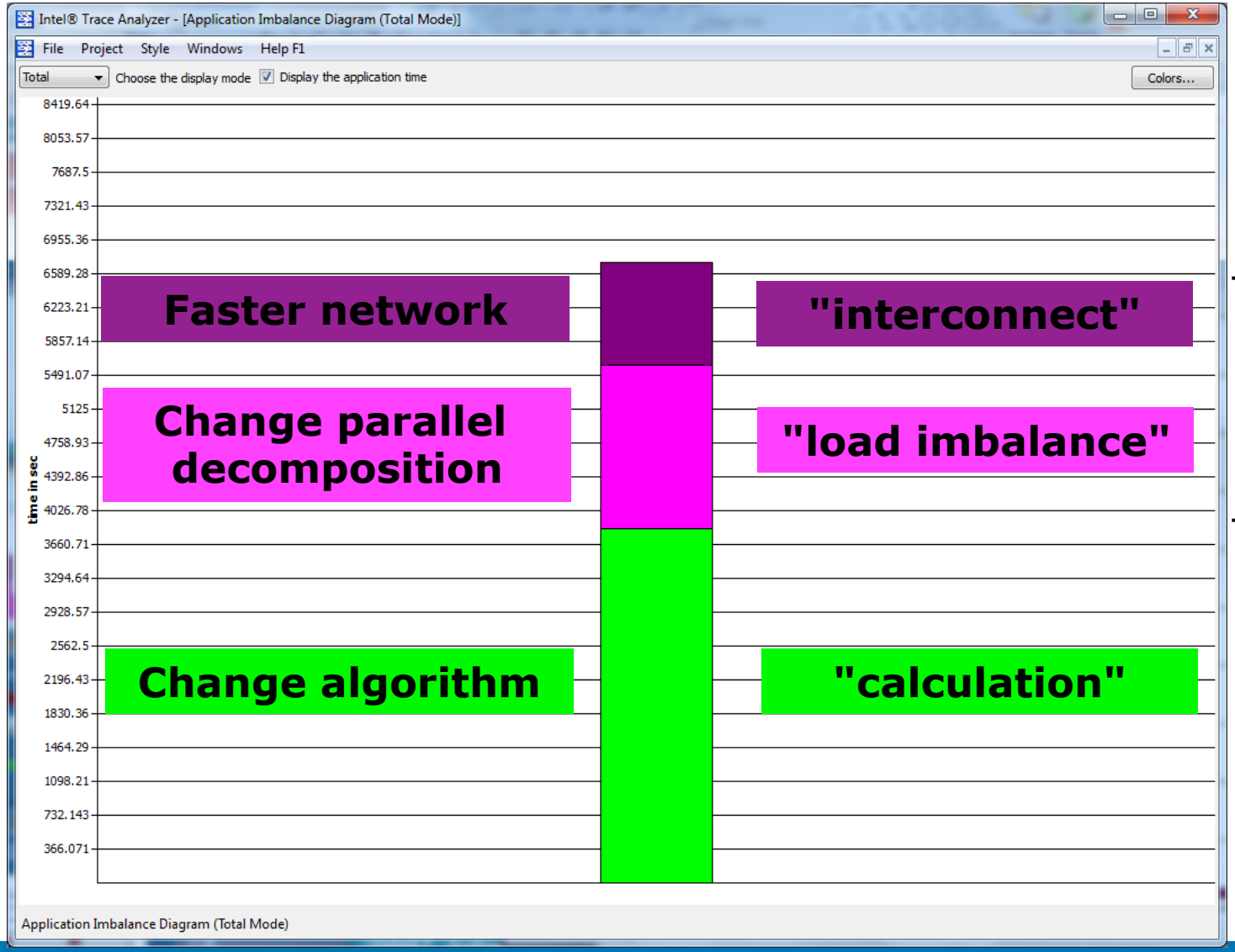
Simulated trace (Ideal interconnect)

Same MPI_Alltoallv

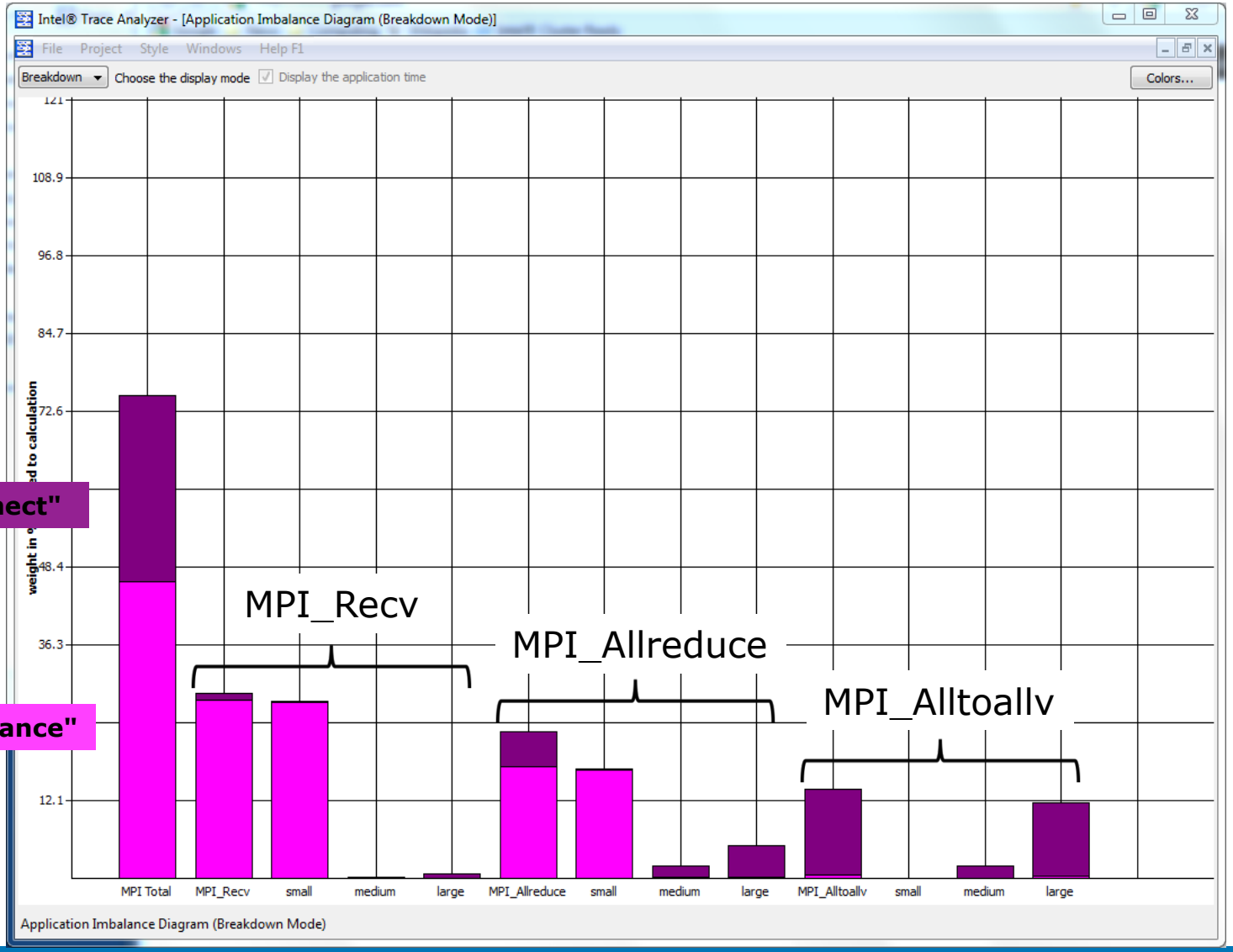
Legend:
257 = MPI_Alltoallv
506 = User_Code

Same timescale in both figures

Application Imbalance Diagram: Total



Application Imbalance Diagram: Breakdown



"interconnect"

"load imbalance"

Outline

- Overview
- Prerequisites of Intel® MPI
- Programming Models
- Hybrid Computing
- Intel® Trace Analyzer and Collector
- Load Balancing
- Debugging

Online Resources

- Intel® MPI Library product page
  www.intel.com/go/mpi
- Intel® Trace Analyzer and Collector product page
  www.intel.com/go/traceanalyzer
- Intel® Clusters and HPC Technology forums
  http://software.intel.com/en-us/forums/intel-clusters-and-hpc-technology/
- Intel® Xeon Phi™ Coprocessor Developer Community
  http://software.intel.com/en-us/mic-developer

Summary

- The ease of use of Intel® MPI and related tools like the Intel Trace Analyzer and Collector extends from the Intel Xeon architecture to the Intel® Xeon Phi™ coprocessor.

Thank you!

Optimization Notice

Optimization Notice

Intel® compilers, associated libraries and associated development tools may include or utilize options that optimize for instruction sets that are available in both Intel® and non-Intel microprocessors (for example SIMD instruction sets), but do not optimize equally for non-Intel microprocessors. In addition, certain compiler options for Intel compilers, including some that are not specific to Intel micro-architecture, are reserved for Intel microprocessors. For a detailed description of Intel compiler options, including the instruction sets and specific microprocessors they implicate, please refer to the "Intel® Compiler User and Reference Guides" under "Compiler Options." Many library routines that are part of Intel® compiler products are more highly optimized for Intel microprocessors than for other microprocessors. While the compilers and libraries in Intel® compiler products offer optimizations for both Intel and Intel-compatible microprocessors, depending on the options you select, your code and other factors, you likely will get extra performance on Intel microprocessors.

Intel® compilers, associated libraries and associated development tools may or may not optimize to the same degree for non-Intel microprocessors for optimizations that are not unique to Intel microprocessors. These optimizations include Intel® Streaming SIMD Extensions 2 (Intel® SSE2), Intel® Streaming SIMD Extensions 3 (Intel® SSE3), and Supplemental Streaming SIMD Extensions 3 (Intel® SSSE3) instruction sets and other optimizations. Intel does not guarantee the availability, functionality, or effectiveness of any optimization on microprocessors not manufactured by Intel. Microprocessor-dependent optimizations in this product are intended for use with Intel microprocessors.

While Intel believes our compilers and libraries are excellent choices to assist in obtaining the best performance on Intel® and non-Intel microprocessors, Intel recommends that you evaluate other compilers and libraries to determine which best meet your requirements. We hope to win your business by striving to offer the best performance of any compiler or library; please let us know if you find we do not.

Notice revision #20101101

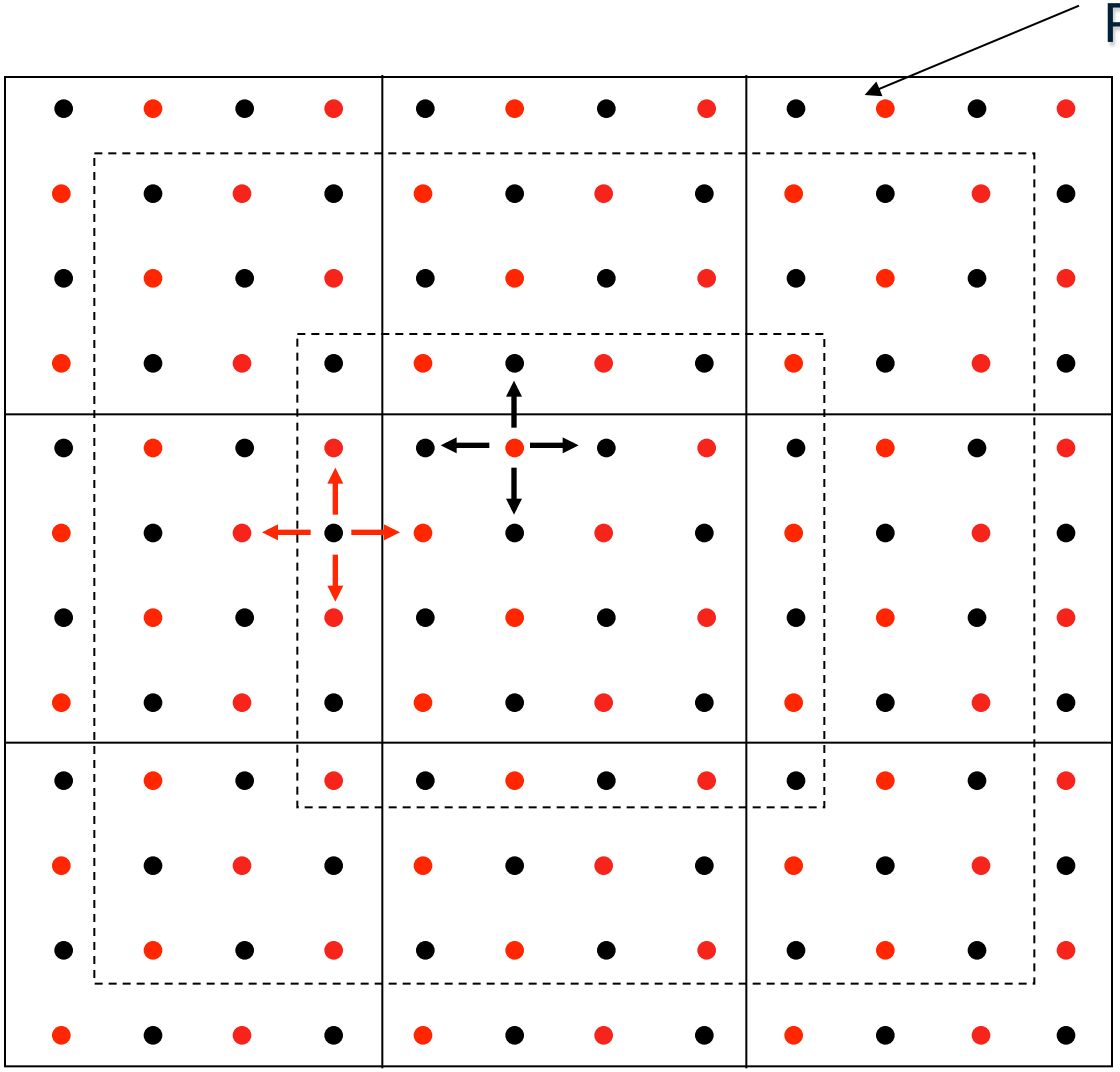
Outline

- Overview
- Installation of Intel® MPI
- Programming Models
- Hybrid Computing
- Intel® Trace Analyzer and Collector
- Load Balancing
- Debugging
- Labs

Labs

- Intel® MPI and ITAC already installed!
- Lab 0: Prerequisites for real labs (executed by user):
  - Upload MPI libraries
  - Set user evnvironment for compiler and MPI
- Lab 1: Basic MPI test
  - Compile and run MPI test.c on Host, Coprocessor, Host+Coprocessor
  - Investigate MPI mapping and OpenMP affinity
- Lab 2: „real" Poisson application, a hybrid MPI/OpenMP code
  - Compile and run poisson on Host, Coprocessor, Host+Coprocessor
  - Vary the number of MPI processes and OpenMP threads
- Lab 4: ITAC analysis
  - Collect MPI trace data for the poisson application and analyze it
  - Try to improve the load balance
- Lab 5: Debug session
  - Debug an MPI programm on Host+MIC
- Lab 3: Hybrid MPI/Cilk Plus (functionality test)
  - Repeat Lab 2 with Intel® Cilk™ Plus
  - Vary the number of MPI processes and Intel® Cilk™ Plus workers

Lab code "Poisson"



red points only access blacks and v.v.; after a relaxation of reds, the red access arrows for blacks partially point to other processes' domain (=> exchange); dashed "halo": process keeps duplicates of the neighbor's boundary points

Lab code "Poisson"

- Solve a Poisson PDE on a discrete square grid
- An N by N array **U** represents the solution; split across parallel processes by rectangular sub-grids
- Main operation: "relaxation" of **U**
- Main parallelization trick: use "red-black" pattern of relaxation
- Means: grid is checkerboard-like divided in red and black points
- In the first big loop over the grid, only red points are updated and each one doesn't access any other red points (fully parallel)
- Then, updated red points have to be communicated (exchanged) along process boundaries
- Then the same for black points, followed by another exchange
- Last, a residual is calculated and eventually the whole **U** collected to a master process

Back up

Debugging Intel® MPI Applications

- Use environment variables:
  - `I_MPI_DEBUG` to set the debug level
  - `I_MPI_DEBUG_OUTPUT` to specify a file for output re-direction
    - Use format strings like *%r*, *%p* or *%h* to add rank, pid or host name to the file name accordingly
- Usage:

```
# export I_MPI_DEBUG=<debug level>
```

or:

```
# mpirun -env I_MPI_DEBUG <debug level>
   -n <# of processes> ./a.out
```

- Processor information utility in Intel® MPI :

```
# cpuinfo
```

  - Aggregates /proc/cpuinfo information

Debug on the Host Using the Intel® Debugger

1. Set up the Intel® Composer XE for Linux for Intel® Xeon Phi™ coprocessor environment:

   ```
   # source <install_dir>/bin/compilervars.sh intel64
   ```

2. Establish environment setting for the Intel® MPI Library:

   ```
   # source <MPI_install_dir>/bin/intel64/mpivars.sh
   ```

3. Compile the MPI source (test.c) for host with debug flag "-g"

   ```
   # mpiicc -g <sourfile> -o <executive-host>
   ```

Debug on the Host Using the Intel® Debugger

4. Start the Intel® debugger idb as the MPI executable and run the real application under debugger control.

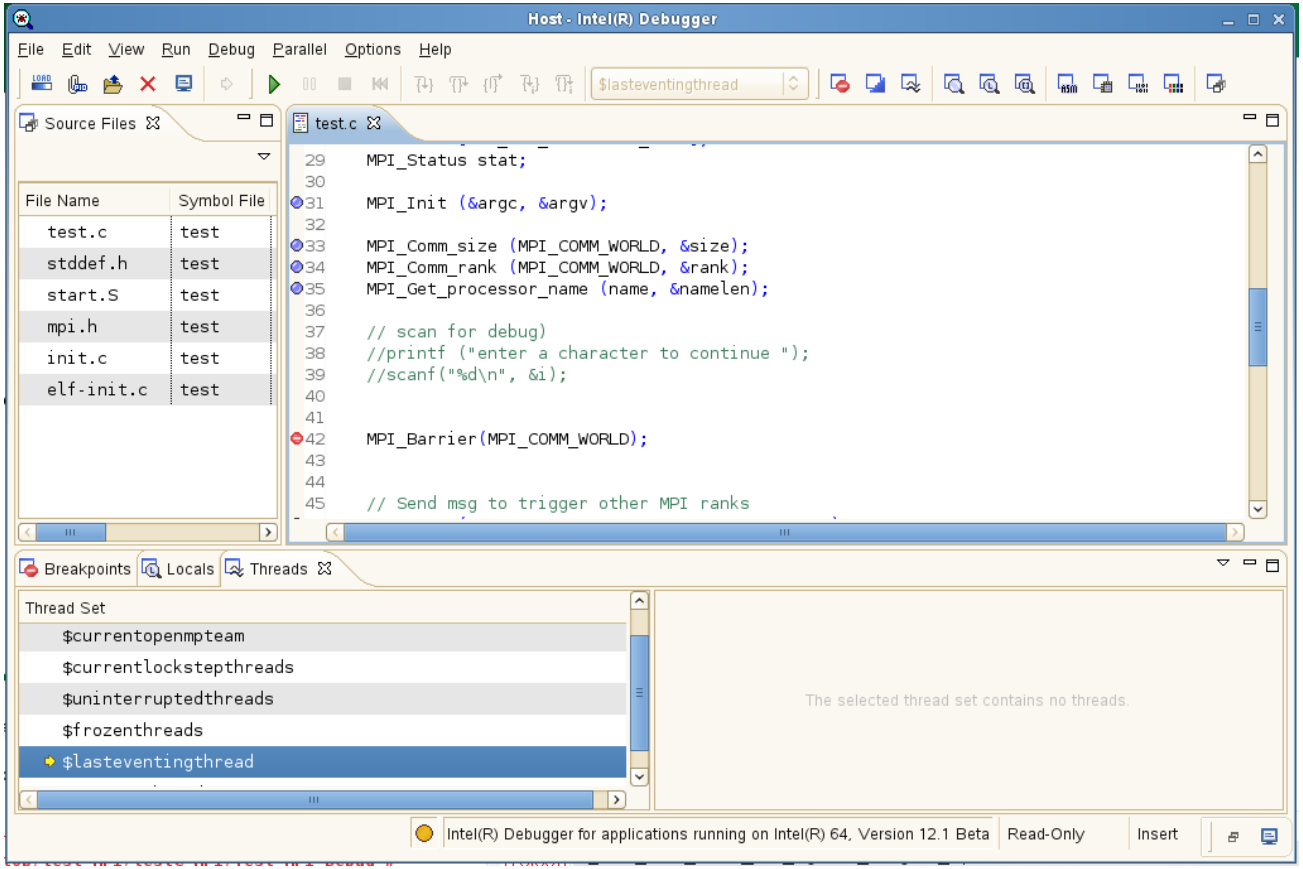
```
# mpirun -n <num> -host <hostname> idb ./<hostexec>
```

Example: Start two debugger sessions for two ranks of the actual MPI program on the host. Each debugger instance controls one MPI process.

```
# mpirun -n 2 -host myhost idb ./test
```

Debug on the Host Using the Intel® Debugger

- In each host debugger, click File-> Open Source File and navigate to the lines that you are interested.
- Insert break point and hit F5 (run).



Debug on the Host and Coprocessor Using the Intel® Debugger

- In one console window (console #1), follow the steps of the preceding section to set up the environment on host
- In another console window (console #2), set up the Intel® Composer XE environment:

```
# source <install_dir>/bin/compilervars.sh intel64
```

- Establish environment setting for the Intel® MPI Library:

```
# source <MPI_install_dir>/bin/intel64/mpivars.sh
```

Debug on the Host and Coprocessor Using the Intel® Debugger

- Compile the MPI program (test.c) for the coprossor with debug flag “-g” (there are some warnings, but that is normal)

```
# mpiicc -mmic -g <source> -o <executive-mic>
```

- No NFS: Upload the executable to the MIC card

```
# scp ./test.mic host-mic0:/tmp/test.mic
```

- No NFS: Upload the MPI debug library to the coprocessor

```
# scp <MPI_install_dir>/mic/lib/libmpi_dbg.so.4.0
                        /lib64/libmpi_dbg.so.4
```

Debug on the Host and Coprocessor Using the Intel® Debugger

- Upload the MPI libraries to the coprocessor (if not mounted or not done before)
- Your application may need additional libraries, for example:

```
# sudo scp <install_dir>/compiler/lib/mic/libimf.so
          host-mic0:/lib64/
# sudo scp <install_dir>/compiler/lib/mic/libsvml.so
          host-mic0:/lib64/
# sudo scp
       <install_dir>/compiler/lib/mic/libintlc.so.5
          host-mic0:/lib64/
```

Debug on the Host and Coprocessor Using the Intel® Debugger

- Now in the console #1, run the debugger to debug the program on host. For example, the following command starts a rank on the host and a rank on the coprocessor. The debugger on the host controls the MPI process on the host:

```
# mpirun -n 1 -host <hostname> idb ./test : -n 1
                -host host-mic0 -wdir /tmp ./test.mic
```

- In the Host Debugger, insert a break point to stop the process and hit F5 for run. The program stops at the break point so we have time to launch the target debugger.

Debug on the Host and Coprocessor Using the Intel® Debugger

- Get the process id of the MPI process on the coprocessor, e.g.:

```
# ssh root@mic0 top -n 1 | grep test
```

- In the console #2, start the Target Intel Commandline Debugger idbc_mic :

```
# idbc_mic -tco -rconnect=tcpip:mic0:2000
```

- Attach to the pid of the MPI program running on the coprocessor:

```
# (idb) attach <pid> target-bin-path
```

e.g.,

```
# (idb) attach 1927 /root/Desktop/debug/test.mic
```

- Type “l” to list the source and “b <line_num>” to insert a break point at <line_num> and run:

```
# (idb) l
# (idb) b 37
```

Thank you!